WARWICK
THE UNIVERSITY OF WARWICK

A Thesis Submitted for the Degree of PhD at the University of Warwick

Permanent WRAP URL:

http://wrap.warwick.ac.uk/178094

Copyright and reuse:
This thesis is made available online and is protected by original copyright.
Please scroll down to view the document itself.
Please refer to the repository record for this item for information to help you to cite it.
Our policy information is available from the repository home page.

For more information, please contact the WRAP Team at: wrap@warwick.ac.uk

warwick.ac.uk/lib-publications

Modular Reasoning About Combining Modular Compiler Phases

Eleanor Davies

A thesis presented for the degree of
Doctor of Philosophy in Computer Science

Department of Computer Science
University of Warwick

March 2021

Supervisor: Dr Sara Kalvala

Contents

Abstract **5**

Acknowledgements **6**

Disclaimer **7**

1 Introduction **8**
1.1 Motivation 10
1.2 Thesis Contributions 13
1.3 Thesis Overview 15

2 Background **17**
2.1 Introduction 17
2.2 Structuring a Compiler 20
2.2.1 Modularity in compilation. 21
2.2.2 Aspect-oriented approaches. 28
2.2.3 Graph structures in compilation. 33
2.2.4 Intermediate languages in compilation. 37
2.3 Reasoning About Compilers 38
2.3.1 The evolution of compiler verification. 38
2.3.2 A variety of proof technologies. 39
2.3.3 CompCert and program simulations. 43
2.3.4 Compositional compiler correctness. 43
2.3.5 Types and compilers. 45
2.4 Modular Reasoning Techniques 46
2.4.1 Hoare-style postconditions. 47
2.4.2 Design by contract. 48
2.4.3 Property based testing. 49
2.4.4 Automated theorem proving. 50
2.4.5 Interactive theorem proving. 50
2.5 Summary 51

3 Related Work **53**
3.1 Miniphase Fusion 54
3.1.1 Motivation for miniphase fusion. 54
3.1.2 The miniphase fusion mechanism. 55
3.1.3 Reasoning about miniphase fusion. 56
3.1.4 Comparisons with the nanopass framework. 58
3.2 Fusing Tree Traversals 59
3.2.1 Exploiting temporal locality. 59
3.2.2 Exploiting static dependence analysis. 60
3.2.3 Exploiting code motion. 60
3.2.4 Exploiting logical reasoning frameworks. 61
3.2.5 Comparisons with miniphase fusion. 61
3.3 Deforestation and Stream Fusion 62
3.3.1 The origins of deforestation. 62
3.3.2 Safety definitions in deforestation. 63
3.3.3 Deforestation and category theory. 63
3.4 Equality Saturation and PEGs 64
3.5 Conclusions: Background and Related Work 65

4 Methodology **67**
4.1 Research Problem 68
4.1.1 What does it mean for compiler phases to be fusible? . . 68
4.1.2 Can we determine whether phases are fusible without attempting to combine them? 69
4.1.3 The scope of the problem. 70
4.1.4 Illustrative Examples 71
4.1.5 The Coq Proof Assistant 73
4.2 The Remaining Chapters 74

5 Fusing Tree Transformations **76**
5.1 What Do We Mean By Successful Fusion? 77
5.2 Defining Transformations & Postconditions 79
5.2.1 Defining tree structures. 79
5.2.2 Defining tree transformations. 80
5.2.3 Defining fusion of tree transformations. 83
5.2.4 Does the order of fusion matter? 86
5.2.5 Defining postcondition checks. 86
5.2.6 Tree transformations and their postconditions. 88
5.3 Fusing Pairs of Transformations 89
5.4 Fusing Three or More Transformations 95
5.5 Fusing Blocks of Fused Transformations 99
5.5.1 Considering blocks as tree transformations. 100
5.5.2 Reusing established properties. 101
5.5.3 How useful is this block-based approach? 101
5.6 Preserving Correctness 102
5.7 Discussion 104

6 Fusing Graph Transformations **106**
6.1 Limitations of the AST Representation 107
6.2 Optimising Loops 108
6.3 Extending the Formalisation 109
6.3.1 Representing graphs inductively. 109
6.3.2 Rebuilding the model. 111
6.4 Illustrative Examples 115
6.4.1 Constant folding 116
6.4.2 Loop induction variable strength reduction 117
6.5 Traversal Order & Inductive Representation 118
6.6 Discussion 119

7 Discussion & Conclusions **121**
7.1 Thesis Contributions 121
7.2 Using Modular Criteria 123
7.2.1 Using a proof assistant. 124
7.2.2 Automated static analysis. 125
7.2.3 Runtime checks. 125
7.2.4 Property-based testing. 125
7.3 Potential Future Work 126
7.3.1 Realistic compiler implementation. 126
7.3.2 Evaluating modularity gains. 127
7.3.3 Evaluating performance gains. 127
7.3.4 Sophisticated compiler correctness. 127
7.4 Thesis Conclusions 128

List of Figures

1.1 A simple AST example. 12
1.2 Applying optimisations separately and in combination. 12

2.1 Code snippet for graph structure examples. 33
2.2 Abstract syntax tree (AST) for code in Figure 2.1. 33
2.3 Directed acyclic graph (DAG) for code in Figure 2.1. 35
2.4 Control flow graph (CFG) for code in Figure 2.1. 36
2.5 Program expression graph (PEG) for code in Figure 2.1. 36

5.1 An AST example. 79
5.2 `arith_fold`: constant folding for given arithmetic expressions on ASTs. 81
5.3 Transforming an AST with `arith_fold` on a postorder traversal. 82
5.4 `if_fold`: constant folding for if expressions on ASTs. 83
5.5 Transforming an AST with `fused arith_fold if_fold`. 84
5.6 `int_to_bool`: A translation from integers to Booleans in ASTs. . 85
5.7 Transforming an AST with `fused int_to_bool if_fold`. 87
5.8 Evaluating a predicate on an AST, both directly and using `check` to check recursively. 88
5.9 Illustrating Fusion Criterion 1 (`FC1`) 90
5.10 Illustrating Fusion Criterion 2 (`FC2`) 90
5.11 Blocks of fused transformations. 99

6.1 A simple graph example. 110
6.2 Loop induction variable strength reduction on pseudocode. . . . 117
6.3 Loop induction variable strength reduction on PEGs. 117

Abstract

Compilers are large and complex pieces of software, which can be challenging to work with. Modularity has significant benefits in such cases: building a complex system from a series of simpler components can make understanding, maintaining, and reasoning about the resulting software more straightforward. Not only does this modularity aid the compiler developer, but the compiler user benefits too, from a compiler that is more likely to be correct and regularly updated. A good focus for modularity in a compiler lies in the phases that make up the compiler pipeline.

Often, compiler phases involve transforming some graph structure, in order to perform program rewriting. Techniques for automatically combining such graph transformations aim to promote modularity whilst mitigating the increased performance overheads that can occur from an increased number of separate transformations. Nevertheless, it is important that the effectiveness and correctness of compiler phases is not compromised in favour of modularity or performance. Therefore, the combined graph transformations need to still satisfy the intended outcomes of their individual components.

Many existing approaches either take an informal approach to soundness, or impose conditions that are too restrictive for the kind of graph transformations found in a realistic compiler. Some approaches only allow transformations to be combined if the ensuing transformation will produce identical results. However, certain compiler optimisations behave more effectively in combination, thus producing a different but better optimised result. Another limitation of some approaches is that, although the compiler phases are intentionally modular, the process of combining them is often tested or reasoned about in a non-modular way, once they have already been combined.

Thus, this thesis outlines an approach for *modular reasoning* about successfully combining modular compiler phases, where success refers to preserving only the truly necessary behaviour of transformations. Focusing on postorder transformations of, first, abstract syntax trees and, then, program expression graphs, the fusion technique of interleaving transformations combines compiler phases, reducing the number of graph traversals required. Postconditions allow compiler developers to encode the behaviour required of a given compiler phase, with preservation of postconditions then a significant part of successful fusion. Building on these ideas, this thesis formalises the idea of postcondition preserving fusion, and presents criteria that are sufficient to facilitate modular reasoning about the success of fusion.

Acknowledgements

There are some invaluable people who have steered me through the process of producing this thesis, and to whom I owe many many thanks.

To my supervisor, Dr Sara Kalvala, for her endless ideas, knowledge, support, encouragement, guidance, and patience - without which I would never even have reached the start line.

To my advisors, Dr Arshad Jhumka and Dr Gihan Mudalige, for stress testing my work, providing a different perspective, and pushing me in new directions.

To members (past, present, and fleeting) of the Warwick DCS community, for their inspiration, distractions and various words of wisdom.

And to my family and friends, for putting up with me through all of it.

Disclaimer

This thesis is submitted to the University of Warwick in support of my application for the degree of Doctor of Philosophy. It has been composed by myself and has not been submitted in any previous application for any degree.

The work presented (including data generated and data analysis) was carried out by the author.

Parts of this thesis have been published by the author as:
"*Postcondition-preserving fusion of postorder tree transformations.*" - Eleanor Davies and Sara Kalvala. Proceedings of the 29th International Conference on Compiler Construction. San Diego. February 2020.

This PhD was funded by the EPSRC Doctoral Training Partnership.

Chapter 1

Introduction

A compiler is generally a large and complex piece of software. Typically, it needs to take code in an expressive, high-level source language and produce an equivalent, executable program in a lower-level target language. This requires many checks and translations, to deal with the high-level language features that assist the developer but are not useful for the machine that the program eventually runs on. Moreover, most modern compilers are optimising, meaning that they seek to produce efficient target code and eliminate common inefficiencies introduced by the developer. Again, this can require intricate analysis and translations of the program being compiled. Therefore, compilers can become confusing and difficult to work with, unless they are well structured.

The process of compilation tends to be split into a sequence of successive phases, each transforming or analysing the program in question and passing the result to the next phase. This typical compiler structure can beneficially exploit modularity, which is the principle of using small, carefully designed components to construct a larger and more complex system. For instance, if each compiler phase performs one task, then it is clear, to the compiler developer, which part of the compiler is performing which part of the compilation. In this way, modularity makes it more straightforward to understand, maintain and debug a complex piece of software, such as a compiler.

There may, however, be a trade-off between modularity and performance. Often there are overheads incurred from combining the separate modules in order to construct the required software. This may be because some work is being repeated across modules that has not been abstracted out as part of the interface. Performance issues may also occur when data structures containing large amounts of information need to be constructed and passed between modules.

Performance is a crucial property of a compiler. Long compilation times can have a detrimental impact on the popularity of a compiler, or even the language that it compiles. It is common practice for developers to compile their code often, checking for compile-time errors and testing as they go. Moreover, integrated development environments are often set to recompile whenever changes to the code-base are made. Thus, a slow compiler can massively reduce developer productivity.

Therefore, in many cases compiler modularity is ignored or sidelined, in favour of efficiency. Under pressure to improve performance, the compiler developer will often choose to combine several unrelated tasks into a single compiler phase. For instance, a phase originally intended to perform a check on the program, without making any changes to it, might be co-opted to also include some program transformations. This kind of ad-hoc manual process can produce an unwieldy monolithic compiler that is difficult to maintain and very unapproachable for new contributors.

One approach to compiler structuring that aims to promote both modularity and performance is miniphase fusion [1, 2]. The miniphase framework imposes a particular structure on all compiler phases, which are specified as transformations on abstract syntax trees (ASTs). There is then an algorithm for automatically fusing these transformations by interleaving them at compiler runtime. This technique reduces the number of AST traversals required to perform the required transformations. Hence, the compiler developer can write and work with modular compiler phases, whilst mitigating some of the impact on performance. The miniphase framework was used to implement the Dotty Scala compiler, which demonstrated the real benefits of such an approach. We use

the miniphase approach as a starting point for much of the work in this thesis.

1.1 Motivation

In addition to modularity and efficiency, another key property of a compiler is correctness. Compiler verification is a costly and intricate process, and bugs in a compiler can have drastic safety implications. Therefore it is important for any compiler technique, such as miniphase fusion, to strongly consider the effect that it has on correctness. Fusing compiler phases must not detrimentally change how they behave.

Petrashko et al. [1] set out an informal set of guidelines as to when miniphases can be successfully fused, relying on developer experience and detailed knowledge of the compiler. These guidelines are augmented by a system of developer-defined postconditions, for individual miniphase, which are checked during testing. Such an approach has the advantage of allowing the compiler developer to convey their understanding and expertise of what a given phase needs to accomplish.

However, it does not take a particularly modular perspective, despite modularity being the initial motivation for the miniphase framework. Rather, all of the postconditions for a block of fused miniphases are checked together, on the result of running the whole fused block. Moreover, relying solely on developer intuition and testing leaves room for problematic corner cases to slip through the net.

Since miniphases revolve around transforming an AST, we can look to the wealth of work on tree transformations to explore other applicable approaches. In particular, there are various related projects that look at combining or fusing tree traversals. Such related work tends to take a more formal reasoning approach, and provide very strong soundness guarantees, such that the result of executing fused tree traversals will be identical to the result of executing them separately.

Soundness means that if traversals are defined and fused according to the

rules of a given framework, then that fusion will always be successful. In other words, those rules are sufficient to ensure successful fusion. The precise definition of successful fusion varies between different frameworks. A popular definition is that: the result of running fused tree traversals must be identical to the same traversals run sequentially. That is, given two tree traversal functions `trav1` and `trav2`, for any tree `t` the following must hold:

$$\texttt{(fused trav1 trav2) t} = \texttt{trav2 (trav1 t)},$$

where `fused trav1 trav2` represents the traversals having been fused within the relevant framework.

The modularity and formal nature of these tree traversal techniques is appealing. However, in the context of considering compiler phases, we tend to find two limitations to such approaches: that they are overly restrictive in terms of the kind of tree transformations that are permitted, and that they are overly cautious when deciding whether fusion is allowed. We discuss those two limitations further in the next paragraphs.

Firstly, it is important to note that, these tree traversal techniques are designed for tree traversals, rather than transformations. This means that they limit the extent of changes that can be made to the tree. For instance, they might only permit changes to the label of the node being visited, and not permit changes to the children of the current node. Such assumptions facilitate strong soundness guarantees, and there are many applications where these kind of traversals are useful. However, compiler phases often have to make widespread changes to the tree structure in order to be effective. So, we need to be able to consider a less restrictive kind of tree transformation.

Secondly, many approaches assume that to be fusible, the fused tree traversals must produce identical results to those from the same traversals run separately. However, for compiler phases such as optimisations, the result of fusion could produce better, and hence not identical, results. That is, optimisations can sometimes be more effective in combination. The following example demonstrates an instance of this.

Example 1. Suppose that we implement two simple AST optimisations, for specific cases of constant folding. Let the optimisation `plus_zero` perform a postorder traversal of the AST, and evaluate all cases of adding 0. And let `mult_zero` also perform a postorder traversal of the AST, and evaluate all cases of multiplying by 0.

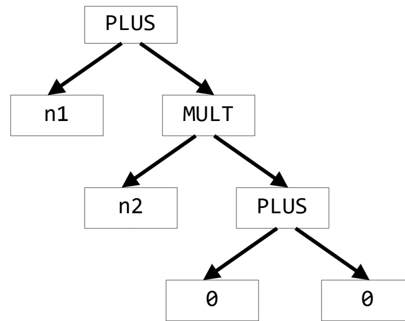

Figure 1.1: A simple AST example.

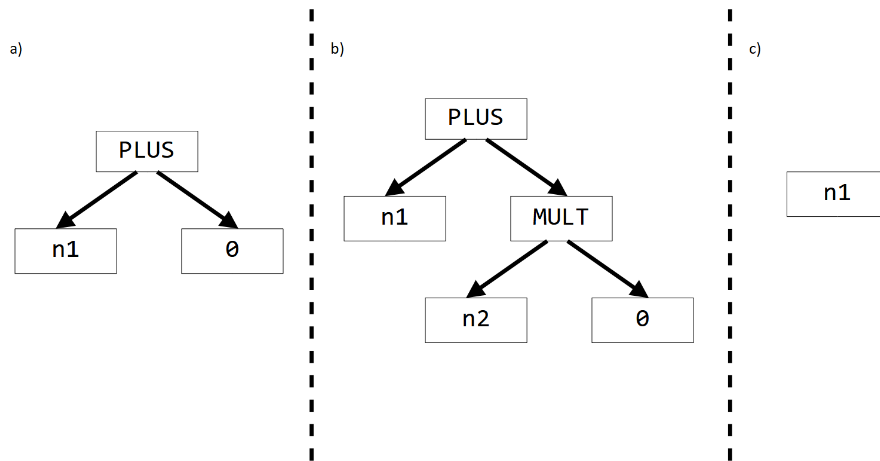

Figure 1.2: Applying optimisations separately and in combination.

Let `t` be the tree represented by Figure 1.1. We can easily see that performing one transformation on the entirety of `t`, before performing the other, results in a tree that could still be optimised further. In Figure 1.2, the tree at a) is the result of applying `plus_zero` and then `mult_zero` to `t`, whilst the tree at b) is

the result of applying them in the opposite order.

However, the tree at c) results from interleaving the two transformations, and having them both try to rewrite each node visited on a single postorder traversal. Fusing these optimisations, in this way, has made the process more effective, since the result cannot be optimised further. So the result is not identical but perhaps we prefer it.

This example is relatively trivial, but it does illustrate the need for a more permissive definition of fusibility. It does not follow that all optimisations are more effective in combination: some combinations will be worse. Nor do we necessarily seek, in this thesis, to determine how fusion would affect effectiveness, in any given case. Rather, by relaxing the requirement for fused transformations to produce identical results, the compiler developer has more freedom to make an informed choice about when fusion is appropriate.

1.2 Thesis Contributions

To exploit fusion of compiler phases to maximum advantage, we need a wider idea of when transformations can be successfully fused. There are cases of transformations that improve with fusion, despite interfering to produce a different outcome when fused. Thus, to fuse as wide a range of transformations as possible, we should be less conservative and strive to include such cases. We need an appropriate definition of fusibility.

Modular Criteria for Postcondition-Preserving Fusion of Tree Transformations

In the case of tree transformations as compiler phases, each transformation is generally intended to establish a property of the resulting AST, such as the absence of a particular language feature. Fusing transformations should not interfere with the ability of a transformation to do such a job. To this end, we can define corresponding postconditions that must hold after a transformation

has run, whether fused or not. This mirrors the approach taken by Petrashko et al. [1], implementing postconditions for Dotty phases to check during testing.

One issue with the Dotty approach is that such testing only takes place on the composite fused tree transformations. This is, perhaps, a convenient place to put postcondition checks, as this testing will take place anyway. However, if we wanted to reason more formally about whether postconditions are preserved by fusion, it would be beneficial to avoid the complexities of the resulting fused transformation. Given that we are aiming to promote modularity in implementation, we should also seek modularity in reasoning and verification.

Thus, we present and verify modular criteria, relating individual tree transformations and postconditions, that are sufficient to guarantee postcondition-preserving fusion. This notion of successful fusion is, in effect, parameterised by the specific postconditions chosen. The criteria are highly modular, in that: adding another transformation and corresponding postcondition to a set of successfully fused tree transformations does not invalidate the established relationships between existing transformations and postconditions. They need only be checked against the new addition, not rechecked against each other.

Extension to Graph Transformations

In addition to tree transformations, compilers often contain more general graph transformations, acting on structures such as control flow graphs (CFGs) or program expression graphs (PEGs). These facilitate useful optimisations, such as constant folding, dead code elimination and loop optimisations. Given that we wish to reap the benefits of transformation fusion as widely as possible, it is advantageous to extend our results about tree transformations to also consider graph transformations.

The main challenge is that, whilst trees naturally lend themselves to inductive definitions which can easily be dealt with recursively, the typical definitions of graph structures look very different. Although a spanning tree can form a tree-like representation of a graph traversal, we still require the extra infor-

mation about edges which it does not traverse. Hence, we look to Erwig's [3] formulation of inductive graphs to aid in this extension.

Using an inductive graph definition, we adapt our work on tree transformations to consider postcondition-preserving fusion of graph transformations. This extension is successful in terms of defining and verifying modular criteria. However, we also discuss the limitations of the inductive definitions, such as the non-deterministic nature of a representation that depends on the unspecified order of construction.

1.3 Thesis Overview

The chapters of this thesis proceed as follows, the first half of the thesis dedicated to setting out context for the work in the latter half.

Chapter 2 details the *background* ideas against which this thesis is set. This splits broadly into three sections: compiler structure, compiler correctness, and more general reasoning techniques. Moreover, the theme of modularity runs throughout these sections.

Chapter 3 surveys the *existing work* that relates more directly to the idea of combining compiler phases or tree transformations, and reasoning about this process. In particular, we examine how modularity, performance, and correctness are approached in such projects.

Chapter 4 outlines and justifies the *methodology* employed in this thesis. This includes specifying the research questions that we are interested in, and setting out the approach that we have taken to exploring those questions. We also discuss some of the tools and techniques used in doing this work.

Chapter 5 sets out a series of definitions and proofs for postcondition-preserving fusion of postorder *tree transformations*. This essentially formalises ideas from the miniphase approach and develops a set of modular criteria for reasoning about fusing tree transformations.

Chapter 6 adapts the work of the previous chapter to consider a more general *graph transformation* definition. To do this we choose an inductive graph

representation, in order to smoothly transition from trees to graphs, and we also discuss the suitability of this choice.

Chapter 7 contains a *discussion* of the success of this thesis in achieving its goals, with respect to its contributions and in comparison with related work. We also outline some potential directions for future work.

Chapter 2

Background

2.1 Introduction

Modularity in computer programming is the idea that large complex programs should be constructed from component parts that are smaller and simpler. This approach has a number of benefits. Modular programs are easier to understand and maintain, as the location of any given task can be narrowed down to its corresponding component. Moreover, testing, formal reasoning, and debugging are all facilitated by the ability to isolate relevant parts and focus in on a given problem. In addition to this, well written program components can be reused in multiple places, reducing code repetition and allowing changes to be made efficiently. Therefore, developers tend to incorporate modularity, be it informally as an aspiration, or using a more formal framework for implementation.

Parnas [4] established the idea of *information hiding* as a key part of modularity, which had previously focused on code reuse as the central objective. The essence of information hiding, or abstraction, is that the user or developer should have exactly the information that they need, that is no more and no less than strictly necessary. Parnas derived an approach for formally and precisely specifying program components, but without including excess information within that specification. In particular, attributes that are likely to change should be

kept secret. This permits internal reimplementations that do not alter how the module has to be interacted with externally.

Parnas, Clements and Weiss [5] were keen for formal modular methodologies to become part of the real-world toolkit, rather than remaining the preserve of academic conference papers. They argued that information hiding allows developers to identify which details of components are vital for them to understand and which are irrelevant. Moreover, if decomposed in a suitable way, the modules themselves should be simple enough to be easily fully understood. Rather than overcomplicating the development process, when appropriately implemented, a modular approach can, in fact, be helpful.

Baldwin and Clark [6] worked to clarify the concept of modularity in computer systems, building on the ideas of Parnas, in order to form a theory of how modularity influences designs and designers. They identified that the crux of modularity was in a hierarchical structure and the relationships between the elements making up a system. The level of independence of the system's modules is significant here. That is, the weaker the links between modules, the greater the degree of modularity. They also argued that, for a design to be successfully modular, it is necessary to derive and implement a set of *design rules*. Resulting from such stipulations, Baldwin and Clark recognised two specific advantages of modularity.

1. If a module can be altered independently of the rest of the system, then the options for updating and improving the system are *multiplied*.

2. If the design rules for the system are sufficient, then design decisions can be *decentralised*, with designers able to make independent and innovative choices within the scope of the rules.

Thus we see that, a major challenge in implementing modularity lies in the task of effectively combining the modular components. In order to work well together, such components should provide an interface that can be relied on, regardless of how that component is implemented. The method for combining these components can then utilise their interfaces, meaning that it remains

unchanged in the case that a component is reimplemented. In addition to this, tasks such as passing data between components can increase the space or time overheads of the software as a whole. Therefore, software components need to be combined carefully, in order to reap the benefits of modularity without compromising in other important areas.

Since modern software design tends to embrace the concept of modularity, the tools and techniques that are used for such projects need to take modularity into consideration. This includes programming languages, methods of reasoning about software, and implementation strategies. As Baldwin and Clark discovered, the key to realistically useful modularity is to establish a framework which will enforce the ways in which modules interact, as well as providing an idea of what a module should look like in a given situation.

For the most part, this thesis explores modular reasoning about modular compilers. To ground and contextualise this research, we look at existing knowledge about the ideas of modularity, compilers, and reasoning about programs. These are broad and well-established concepts, and so we focus on: defining them, outlining their evolution and current position, and exploring the ways in which they overlap. Most importantly, we can then use this background knowledge to motivate the general problem area of our research. The three main topics covered in this chapter are as follows.

Compiler Structure First, we look at how compilers can be structured, with a focus on their inherent modularity. As complex programs, there are obvious benefits to be gained from a modular approach. Fortunately, the compilation process can generally be decomposed along intuitive lines, since the phases of the compiler pipeline provide a convenient basis for modular components. Moreover, given the wide range of possible source and target languages, along with the variety of potential optimisations, there is a real necessity for interchangeable components that facilitate code reuse.

Compiler Correctness Next, we move on to techniques for reasoning about compilers and, more specifically, the field of compiler correctness. This has been researched both intensively and extensively, since the compiler is an integral part of the software development pipeline. It is vital that target code behaves as specified, otherwise it is not useful. Most formal reasoning is done on source-level code so, in order for it to be worthwhile, the compiler must be known to preserve whatever guarantees are established. However, the process of compiler verification is complex and costly, so effective techniques, exploiting properties such as modularity, are crucial.

Reasoning Techniques Finally, we cover more general reasoning techniques that have a focus on modularity. Many approaches to software correctness are based on or inspired by Hoare-style logic, using preconditions and postconditions to specify program properties. This lends itself to a modular perspective because such conditions naturally form interfaces between segments of code. Formally verifying, or even exhaustively testing, real-world software can be impractical due to its complexity. Modularity can allow the problem to be broken down into related segments that are each of a more manageable size.

Together these background ideas set the scene for the research in this thesis. There is much to be gained from a modular and rigorously formal approach, but this has to be applicable to real-world situations to have an impact. This is a theme that runs through all three of these fields. As we will see, compilers are complex programs for which correctness is paramount, therefore modular approaches to implementation and reasoning are particularly valuable.

2.2 Structuring a Compiler

A *compiler* takes a program written in some source language, and translates it into a target language. Generally, the source language is high-level and expressive, allowing a developer to write effective code, whereas the target language is

comparatively low-level and easily executable. During translation, the compiler will check the program and is able to report certain errors. Many compilers also perform optimisations, so that the efficiency of the emitted target program is improved.

2.2.1 Modularity in compilation.

The need for compilers arose, primarily, from the development of higher-level languages, which make programming more intuitive and approachable for the programmer. The further abstracted these languages are, from the level of machine code, the more potential work a compiler will have and the more complex it will become. Thus, it is vital that the structure of the compiler is clear and organised, for instance via modularity.

In a 1962 history of the compiler [7], Knuth wrote: *"Five years ago it was very difficult to explain the internal mechanisms of a compiler, since the various phases of translation were jumbled together into a huge sprawling algorithm. The passing of time has shown how to distinguish the various components of this process, revealing a basic simplicity."* This suggests that, even relatively early in compiler evolution, some kind of modularity had been reached and identified as an important feature.

Modern compilers lend themselves naturally to a modular structure, since they perform a conceptually modular process. From analysing and representing the input program to generating the output code, the compiler pipeline is naturally divided into a series of separate phases, which can be generalised as follows.

1. Lexical analysis: eliminates unnecessary program elements like certain whitespace and comments, and translates other elements into tokens that can be recognised by the rest of the compiler.

2. Syntax analysis: checks the generated tokens against a specified syntax for the source language, and constructs structural representations of the program, such as parse trees and abstract syntax trees.

3. Semantic analysis: checks that the program is consistent with a specified semantics of the source language, often examining static types.

4. Intermediate code generation: uses the AST to produce code in an intermediate language, that is a lower-level language than the source which is still independent from the specific target architecture.

5. Optimisations: performs general optimisations on the intermediate language, such as constant propagation and optimisations on loops, which will ultimately result in more efficient target code.

6. Target code generation: uses the optimised intermediate code to produce corresponding code in the prescribed target language.

These are common steps in compilation, although some, such as optimisations, are not essential. Regardless of the languages in question, there are various standard techniques for each stage which most compilers will exploit. This means that it is beneficial to implement independent components that can be combined and specialised to produce a compiler for the required language combination.

Generally, a compiler *phase* refers to a conceptual segment of the pipeline, whereas a compiler *pass* is a traversal of the source code to perform some analysis or rewrite. Sometimes this entire compilation process can be completed in a single pass of the source program, which is done in an attempt to improve memory usage and compilation time. Pascal [8, 9], for example, is a notable case of a language that lends itself to single-pass compilation. However, multi-pass compilers are increasingly well supported by modern architectures and continue to be popular due to an increased preference for modularity in general, and through the influence of endeavours such as compiler verification. Moreover, effective optimisations tend to require multiple passes, especially where different optimisations are working together.

One way of imposing modularity on compilers is through the implementation of compiler frameworks. The LLVM (Low Level Virtual Machine) [10] is one extensively used compiler framework, which facilitates modularity. It provides a

low-level type system that is independent of source language. This means that sophisticated analysis techniques, such as dependence analysis, can be implemented once and then reused in cases of different source languages. Moreover, introducing high-level type information to a lower-level representation allows more transformations to be performed at the same level. Thus, components can be interchanged in a way that a multi-level approach does not permit.

Some compilers put modularity even more explicitly at the forefront of their implementation. For example, the miniphase [1, 2] and nanopass [11, 12] frameworks, both encourage small simple compiler phases which are combined in a standardised way. Here, we outline these ideas, exploring how, and how effectively, each approach deals with the challenges of promoting modularity. In the next chapter, we will return to these frameworks, comparing the two approaches and exploring further how they are relevant to the work in this thesis.

The miniphase framework.

Motivation Scala is a high level, statically typed object-functional language. The original intention of the development of Scala was to encourage the constriction of modular software and reusable components. The initial hypotheses [13] for the creation of Scala were twofold:

1. *"a programming language for component software needs to be scalable"*
2. *"scalable support for components can be provided by a programming language which unifies and generalizes object-oriented and functional programming"*

Hence, developments in Scala have been instrumental in perpetuating the combination of features from both paradigms, throughout the landscape of programming language design, and promoting a modular approach to programming.

The miniphase framework [1, 2] was created as part of the experimental Dotty compiler for Scala. The purpose of Dotty was as a platform for trying out new ideas, which could be implemented as a subsequent official compiler

for the launch of Scala 3. In particular, Dotty embraced the idea of having a deliberately modular structure, with large numbers of simple phases, each implementing a specific function, being used to build the compiler pipeline. Here we specifically discuss the version of Dotty outlined in the paper [1] as a stable and simplified case study.

Prior to Dotty and the introduction of miniphases, the standard Scala compiler, scalac, was deemed to be overly difficult to understand, perhaps akin to the jumbled sprawl that Knuth [7] complained of some fifty years earlier. There were phases that were nominally determined to perform one task, but also included some other unrelated tasks. Dotty intended to fix this problem by imposing modularity as an inherent feature of its compiler phases. This is particularly significant in an open-source setting such as Scala, where disparate developers may be working on the language or related tools without comprehensive communication.

Framework The miniphase framework provides a template for what a compiler phase should look like. In essence, a miniphase is a compiler phase which does the following.

- utilise a postorder traversal of the abstract syntax tree
- implement a transform method for each tree node type, and a general transform method that selects from among these the appropriate transformation for any given node
- list the other miniphases that this one must run after, on either a node-by-node or entire tree traversal basis
- specify a predicate over tree nodes that encodes and checks a postcondition

By standardising the structure of the compiler phases, it is easier to determine what an unfamiliar phase is supposed to be doing. Furthermore, whilst the first two of these stipulations pertain to the functionality of the individual miniphase,

the latter two are used to ensure that miniphases relate to each other correctly. So, by design, miniphases draw on key principles of modularity.

Development and Challenges The Dotty compiler is written in Scala itself, exploiting the object oriented nature of the language to enforce modularity. The object oriented programming paradigm has modular ideas, such as encapsulation, at its core. Encapsulation is a strong form of information hiding. For example, certain members of a class may be only accessible via given methods, allowing the class developer strict control over what happens to those members, as well as the freedom to reimplement the class with minimal effect on its users. Dotty uses Scala classes to implement the trees and miniphases that it requires, exploiting inheritance to achieve both code reuse and uniform phase structure.

The predominant challenge that the miniphase approach sought to overcome was that of performance. By modularising, and thus increasing the number of, AST transformations, the AST needed traversing many more times. Similar performance issues had been addressed previously, in scalac, by manually combining the source code of different transformations, and thus losing any sense of modularity. In attempt to improve on this, the miniphase framework provides a fusion algorithm for automatically combining miniphase transformations before they are run. A standardised fusion method is made possible by the strict conditions imposed on how miniphases are implemented and produces a performant compiler which is still inherently modular. The motivation and implementation of this miniphase fusion are discussed in further detail in Section 3.1.

Another challenge facing miniphases was in ensuring that the restrictions on compiler phases are not overly restrictive. The purpose of standardising parts of miniphases such as the traversal order is to allow them to be automatically fused, as well as improving clarity. However, the required stages of compilation still all need to be performed. The Dotty project demonstrated the viability of a realistic compiler that adheres to this framework, and showed that phases that previously violated the miniphase specification could be reimplemented within it.

Uses and Benefits The version of Dotty outlined in the paper [1] had 54 miniphases, with stated intentions to at least double this number. Together the phases were sufficient to compile Scala source code down to Java bytecode. These Dotty miniphases included: analyses and checks on features like abstract and static members, translations of features like pattern matching and exception catch cases, and optimisations such as tail recursion elimination and method call inlining.

The Dotty team also reported that the introduction of miniphases was helpful to its developers. The simplified and standardised nature of these phases allowed new contributors to get started more quickly. Moreover, it was easier to locate and solve specific problems, without needing to understand a lot of unrelated surrounding code. In addition to this, the preconditions and postconditions that miniphases specify provided a loosely coupled mechanism for testing the work of loosely connected developers.

The nanopass framework.

Motivation Modularity is often exploited for educational purposes, particularly when learning to implement a complex process such as compilation. Modular phases provide learners with the ability to break the task into manageable chunks, and weakly linked components allow a series of assessments to be graded independently of one another. This was a concern of Aiken's Cool compiler project [14], designed for teaching compiler construction and revolving around the purpose-built Classroom Object-Oriented Language. Cool was published, not only to share resources but, to encourage others to share educational resources too.

Indeed, the initial nanopass paper [11] was another example of sharing teaching experience and expertise, and similarly championed a modular approach. The educational nanopass compiler arose from compiler courses at Indiana University. The goal was to present a compiler as made up of small phases, aligning its implementation with the conceptual process of compilation. Moreover, the

framework aimed to provide tools to support the development of this kind of compiler, especially for inexperienced developers.

Framework The nanopass framework standardises the structure of compiler phases, with common processes such as program traversal abstracted into a more general separate algorithm. Each nanopass must provide formally specified input and output languages, along with mappings from input language features to output language features. These intermediate languages may be very closely related, in the case that very little is altered. So, the framework allows one language to extend another existing language, inheriting everything that does not change. This again reduces the amount of code being repeated in writing compiler phases.

Development and Challenges Implemented in Scheme, the nanopass framework takes advantage of the macro system to write tools as extensions. This is significant for achieving code reuse when defining nanopasses and the corresponding intermediate languages, providing the ability to automatically complete certain boilerplate code.

One challenge for the nanopass framework was related to the performance impact of combining many small compiler phases. The concern was that a slow compiler would disillusion the students learning how to write it. An advantage of the modular nanopass approach is that it makes it easier to switch off given phases, such as particularly time consuming optimisations. This not only allows the student to trivially speed up their compiler if wanted but also gives them some intuition of which phases impact heavily on performance. However, performance is not as significant in this case as it would be for a commercial compiler, and hence is not as high a priority.

Uses and Benefits The original nanopass framework was used by students to practice implementing a compiler from Scheme to Sparc assembly code, via over 50 nanopasses. These nanopass phases included: checks such as on the

uniqueness of bound variables, translations of features not present in the output language such as converting from basic blocks to linear stream of instructions, and optimisations such as replacing direct lambda calls with let expressions. The move away from the monolithic compiler structure allowed students to easily implement more complex compilation, and the nanopass framework helped to clarify the educationally meaningful parts of each phase.

Further to this, the nanopass framework has also been used in industrial compiler construction. Keep and Dybvig [12] took the original educational framework as a prototype and built on it, for use in the commercial Chez Scheme compiler, modularising the original 10 phases into around 50 nanopasses. They improved the usability of the framework, adding features that were specifically useful to the real-word developer, and put more emphases on performance. Using the nanopass framework, and including a more expensive register allocation algorithm, the runtime of the implemented compiler was slower than, but within a factor of two of, the previous iteration of the Chez Scheme compiler. It also produced target code which was up to 27% faster than previously produced.

2.2.2 Aspect-oriented approaches.

Another approach to developing modular software, that has been adopted by some compiler projects, is aspect-oriented programming. These aspect-oriented approaches focus on automatically weaving together modular components, a similar idea to that at the heart of miniphase fusion. Whilst miniphases are focused on mitigating the performance impact of modularity, aspect-oriented approaches tend to focus on maintainability, reducing the likelihood for developers to introduce errors when writing compiler phases.

What is aspect-oriented programming?

Aspect-oriented programming centres around the idea of cross-cutting concerns, that is, an area of functionality which is scattered across a program making it difficult to cleanly separate out into a meaningful modular component. For

example, security is a concern that necessarily touches many parts of a program. Suppose, in a business application, that some user security clearance level must be checked before any method can run. Then such an authorisation check must be implemented at each method that is written.

The aim of aspect-oriented programming, then, is to help the developer to effectively separate cross-cutting concerns. Aspects represent cross-cutting concerns, and aspect-oriented languages allow aspects to be defined and then woven together to construct a program. In this way, modularity can introduced to situations where it would traditionally be difficult to achieve.

Aspect-oriented programming is not as widespread or popular as paradigms such as object-oriented programming. It can involve a steep learning curve for developers who are unfamiliar with effectively using aspects. In addition to this, mature tooling to support aspect-oriented programming is not always available. Hence, the separation of cross-cutting concerns must be deemed important enough to warrant a potentially substantial time investment.

One disadvantage of aspect-oriented approaches, with respect to modularity, is that it can negatively impact the readability of code. A touted benefit of modularity is that it makes software easier to understand and maintain. However, the way that aspect-oriented languages weave together elements of different aspects to produce the final program can obscure the control flow. This makes it difficult to follow what is happening by just looking at source code.

Thus far we have explored compiler modularity in terms of compiler phases, that is, in terms of the transformations that must be applied to the program being compiled. However, there is another dimension to consider. The program itself can be viewed as the composition of various syntactic elements, such as variables, constants, and operators. These elements are often represented in compilation as the nodes of an abstract syntax tree (AST), with each transformation defined to rewrite some of these nodes. However, it can be difficult to achieve compiler modularity in terms of both transformations and AST nodes, due to cross-cutting concerns. Aspect-oriented approaches to compiler construction aim to address these kind of difficulties [15].

Inheritance patterns vs. visitor patterns in compilers.

A pattern, particularly in object-oriented programming, is a systematic approach to a common design problem [16]. For instance, inheritance and visitor patterns are popular approaches to designing modular compilers [17]. Inheritance patterns define a super-class for AST nodes, from which each type of AST node inherits and implements the required AST transformations. On the other hand, visitor patterns define transformations that each take account of all possible node types. Inheritance and visitor patterns view modularity in terms of AST node types and AST transformations, respectively, but have difficulty in doing both. Thus, modularity in compilers often occurs in one of these two dimensions. This is a problem of cross-cutting concerns, that can be helped by an aspect-oriented approach.

Inheritance patterns rely on defining a super-class or super-type for AST nodes, which all nodes must inherit from. Such a super-class sets out the transformations that every node must implement, that is, each of the AST translations and optimisations that the compiler performs. Then each node type specifies how each transformation acts on it. In this way, adding a new type of AST node is easily done with minimal alteration to the existing nodes. The new node simply inherits and implements all required transformations for itself. However, if a new transformation must be added, then each node must be altered to add the appropriate implementation of the new transformation on that node.

Visitor patterns, conversely, view modularity from the perspective of the transformations. A super-class is defined for transformations, setting out the node types that each transformation must be able to deal with. For instance, if there are node types `A`, `B`, and `C`, then each transformation would have to implement `transformA`, `transformB`, and `transformC`, or some similarly named methods. This makes it very easy to add a new transformation, since it just needs to inherit from the super-class and implement the required methods. However, it is now more difficult to add a new node type, since each transformation

must be altered to accommodate it.

The conflict between modularity in terms of AST node types and AST transformation, as highlighted by comparing inheritance and visitor design patterns, is a classic example of cross-cutting concerns, which are difficult to untangle. Aspect-oriented programming, then, is an obvious contender for addressing such a problem.

TreeCC: an aspect-oriented compiler-compiler.

One example of an aspect-oriented compiler project is Tree Compiler-Compiler, or TreeCC [17]. As a solution to the competing problems of frequently changing AST nodes and frequently changing AST transformations, TreeCC leans on aspect-oriented ideas. Building on a visitor pattern, which modularises in terms of transformations, aspects are used to also allow the encapsulation of node types. TreeCC then includes a domain-specific language for defining node types and transformations and weaving them together, in an aspect-oriented manner.

A benefit of the TreeCC framework is that the implementations of transformations for different node types can be scattered throughout the codebase. These definitions are then gathered together when TreeCC builds the compiler. This means that when a new AST node is added, the new resulting transformation cases can be implemented in a separate file, rather than having to alter each original transformation individually. Another key element of TreeCC is a mechanism for checking that each transformation considers every defined type of node. This is something that could easily be forgotten when adding a new node type, but will be flagged automatically, avoiding any resulting bugs.

JastAdd: aspect-oriented compiler construction with Java.

Another project that takes an aspect-oriented approach to compiler construction is JastAdd [18]. Built in Java, the JastAdd framework revolves around transformations of an object-oriented AST structure. This object-oriented approach to defining ASTs naturally modularises in terms of node type. Inheritance is

central to object-oriented programming, and so it would be easy to fall into the inheritance pattern approach to compiler construction.

In order to also allow modularity in terms of AST transformations, JastAdd uses a method inspired by inter-type declarations in AspectJ [19], an aspect-oriented Java extension. These inter-type declarations allow additional methods or fields to be declared from outside of the type declaration that they belong to. In JastAdd, different parts of compiler functionality, such as typechecking and code generation, are defined as aspects. Each aspect has its own file or module, containing the relevant methods and fields for each AST node type. The JastAdd system then collects these definitions and automatically weaves them into complete AST node classes.

One difference between the JastAdd and TreeCC frameworks is that JastAdd is specifically designed to be implemented in object-oriented language. This means that it can directly exploit the benefits that object-oriented programming provides when writing modular software. In particular, encapsulation is a key pillar of the object-oriented paradigm, and is also tied to some of the key features of modularity. Encapsulation is the principle of restricting access to the components and implementation of an object. This promotes information hiding and necessitates a well defined interface for objects to interact with each other, which are two important elements of modular software.

Whilst JastAdd borrows the concept of inter-type declarations, from AspectJ, there are many other parts of aspect-oriented programming that it does not take on. Wu et al. [20] explore how AspectJ can be used more directly to facilitate compiler construction. For instance, the join point model of AspectJ allows aspects to define functionality that is inserted into the control flow dynamically, rather than just weaving aspects together statically. This can be used to encode more elements of the compiler, such as the AST traversal order.

2.2.3 Graph structures in compilation.

A compiler that is constructed from a series of separate phases requires a mechanism for passing information from one phase to another throughout program compilation. To this end, graph structures are commonly used to represent programs as they travel along the compiler pipeline. Some compilers use the same representation throughout, whilst others use different forms of graph structure for different phases. The most commonly found examples, in practice, include abstract syntax trees and control flow graphs, although compilers exploit graphs to represent and manipulate programs in a variety of ways [21].

To illustrate each type of graph structure, we provide an example, in the form of a corresponding graphical representation of the following pseudocode snippet.

```
i := 0
while (i <= 10) {
    i := i + 1
}
```

Figure 2.1: Code snippet for graph structure examples.

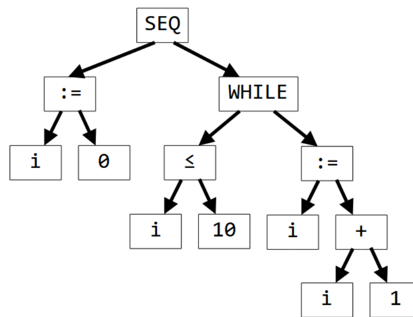

Figure 2.2: Abstract syntax tree (AST) for code in Figure 2.1.

Abstract syntax trees.

Generally used in the early parts of compilation, *abstract syntax trees* (ASTs) represent the structure of a given program. An AST contains only necessary program elements, so it is abstracted from the concrete syntax of the source code. The inner nodes represent operators with the node's children representing its operands. Moreover, since the tree structure itself can encode the hierarchical nature of a program, certain delimiters, such as parentheses and semicolons, may be removed. Thus the AST is more compact than a parse tree which represents the syntax of the given source program more comprehensively.

The AST is often annotated with extra information that the compiler gains from its analysis phases and makes use of later. Various static analyses can be performed on the AST as part of compilation. For example, the semantic analysis phase usually involves static typechecking on the AST. Such typechecking is a compositional process, according to the nature of type inference rules. The AST structure facilitates the decomposition of expressions into subexpressions for this purpose.

Typically, once the frontend analyses have been satisfactorily completed, the AST can be used to generate either intermediate code, to be optimised, or target code. This process involves flattening the tree structure into a more linear form, such as three address code or a different graph based structure.

The Dotty compiler [1, 2], as discussed previously, uses an AST structure throughout its pipeline of miniphases. Such consistency helps the miniphase approach to promote modularity. Since all miniphases are AST transformations, they can easily be interchanged and reused. This would not be the case if different phases used different intermediate representations.

Directed acyclic graphs.

It is possible to avoid repeating identical subtrees, as happens in ASTs, by allowing nodes to have multiple parents, that is by forming *directed acyclic graphs* (DAGs) instead of trees. For example, in the AST in Figure 2.2 there

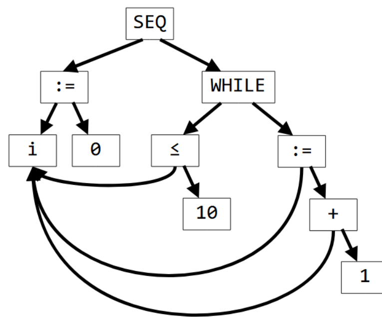

Figure 2.3: Directed acyclic graph (DAG) for code in Figure 2.1.

are multiple nodes representing the variable `i`. In Figure 2.3, the DAG only has one node for `i`, which is shared as required. This DAG representation is yet more compact than the AST, reducing further the burden on memory and the number of nodes to traverse. Moreover, the compiler then has information encoded in the intermediate representation that it can exploit for optimisation.

Identifying shared nodes through constructing this DAG representation can lead to local optimisations such as common subexpression elimination and dead code elimination. Such transformations get rid of unnecessary instructions that are performing computation which has been performed previously or that will never be reached in the program execution. The DAG also facilitates the movement of statements within a block, by highlighting which statements are independent and can therefore be reordered. Essentially, DAGs can represent more information about dependences in the program than ASTs, which can be useful for compilation.

Control flow graphs.

Control flow graphs (CFGs) [22] are commonly used to represent programs at a low-level within the compiler pipeline. Unlike the DAGs discussed previously, CFGs typically need to contain cycles, in order to explicitly represent the control flow of a program, that is the order in which instructions should be executed. To this end, the nodes of a CFG correspond to the program's instructions or

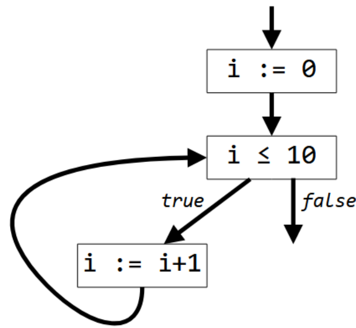

Figure 2.4: Control flow graph (CFG) for code in Figure 2.1.

basic blocks, rather than representing its various syntactic elements.

The analysis of control flow is key to many low-level but architecture independent optimisations. For instance, any parts of the graph, and thus the program, which are never touched by control flow will play no part in program execution. Where this can be determined statically, they can therefore be removed from the compiled code. Moreover, examining control flow facilitates the discovery of loops which naturally provide fruitful optimisation targets. Loops tend to dominate computation time, and thus this is a significant endeavour.

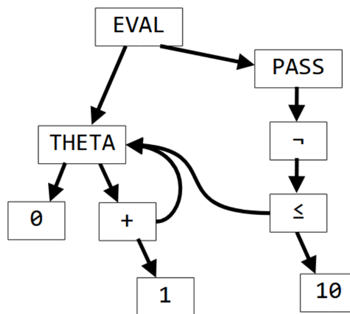

Figure 2.5: Program expression graph (PEG) for code in Figure 2.1.

Program expression graphs.

Another graphical representation for compiling programs is the program expression graph (PEG) [23]. This falls somewhere between the AST and the CFG,

in terms of how it represents programs and how it is typically used. The inner nodes of a PEG represent operators with the children of the node providing the necessary arguments. These operators can include special built-in nodes which have been defined to encode conditionals and loops. Thus, it is easier to identify and manipulate such features in PEGs than for the ASTs and DAGs previously discussed.

Figure 2.5 shows a PEG representation of our code snippet example. The `THETA` node is used to represent loop variables; in this case it represents `i`. The first child of `THETA` tells us that it is initialised to 0, and the second child tells us what happens on each loop iteration. The `EVAL` and `PASS` nodes are used to determine the value of `THETA` at the end of the loop. The descendants of `PASS` encode the terminating condition of the loop, denoting that the loop stops executing if the while loop condition is not met.

A PEG can be obtained from a CFG, and vice versa, with a natural correspondence between the two [24]. One notable feature is that making local PEG changes can have wide-reaching and complex effects on the analogous CFG. For instance, a series of PEG transformations, applying simple equations to perform local rewrites like operator distribution and constant folding, can result in a corresponding CFG with a vastly different node configuration than it had before.

2.2.4 Intermediate languages in compilation.

While the various graph structures used in compilation provide different forms of intermediate representation, compilers can also use the idea of intermediate languages to formally characterise sections of pipeline. Like an intermediate representation, an intermediate language provides a stepping stone in between the source language and target language. This means that part of the compiler is not wholly dependent on these particular languages. It may be that code in the intermediate language is obtained from source code and immediately translated into target code, in which case it simply provides a common point for different

combinations of frontends and backends to be joined together. Alternatively, it may facilitate a series of optimisations which can be easily reused.

As we saw previously, formally specified intermediate languages are a key part of the nanopass framework [11, 12]. In this case, there is not just one intermediate language but many of them. Each language is a slight perturbation on the last, starting from the source language and getting incrementally closer to the target language. This approach precisely specifies the outcome of each compiler phase, with many intentions and invariants able to be encode within the definition of a language. It also facilitates compiler verification, by formally specifying semantics at each level of the compiler which can then be formally related to each other.

2.3 Reasoning About Compilers

It is important to be able to reason about compilers, since their correctness is often paramount. Where bugs are introduced into code, by incorrect compilation, the safety and security of that code can be compromised. As compilers are large and complex pieces of software, it can be difficult to identify or locate the source of such problems. Moreover, because testing may miss obscure problematic cases, formal reasoning techniques provide stronger correctness assurances.

A *correct* compiler should preserve the observational behaviour of any program it compiles. In other words, the target program must behave in the same way as the source program would be expected to. There are various formal definitions of compiler correctness which differ between projects.

2.3.1 The evolution of compiler verification.

Compiler verification is a key component in effectively applying formal methods to software. It is much easier to reason about a program at the source language level than, after it has been compiled, at the target language level. The source language has been chosen, in part, because it is human-readable and allows the

programmer to express and understand what the program should do. This is generally not the case for the lower-level language that it will be compiled to. Otherwise, the program could be written directly in the target language and there would be no need for a compiler. However, if verification is done at the source language level, it is vital to ensure that proven guarantees are preserved by compilation.

Compiler verification is a well established endeavour, first proposed in 1967 [25] and keenly studied since [26]. The main idea is to logically link the source and target versions of the program being compiled. This means that verification of source code is sufficient to provide guarantees about the compiled target code, as well as more informally preserving the programmer's expectations about how their program will behave. Developments such as CompCert [27] have shown that compiler verification is a realistic goal and results in actual benefits, with respect to compiler bugs [28].

2.3.2 A variety of proof technologies.

Appropriate tooling is vital in facilitating the complex task of compiler verification. In general, theorem provers are preferred over solely using automated processes such as model checking. This is predominantly due to the large and complex nature of the compiler as a piece of software. Due to the size of the state space involved, model checking would have to consider a vast number of cases, in order to provide strong correctness guarantees. Thus model checking is an unpopular choice for real-world compiler verification, since memory constraints become problematic.

Early work used the LCF theorem prover for mechanising a simple example [29]. This kind of approach has continued, with the Coq theorem prover used in projects such as CompCert and Vellvm [30], and HOL4 used in CakeML [31]. However, compiler verification still signifies a significant proof effort. For instance, CompCert required 35 000 lines of code, 87% of which corresponded to specification and proof.

Boyer-Moore, Nqthm and ACL2 A number of early projects used the Boyer-Moore or Nqthm theorem prover, such as the CLI verified stack project [32]. This looked to verify a Micro-Gypsy to gate level stack [33], building on a verified translation from Piton assembly language to FM8502 microprocessor [34]. Projects like these recognised the value of correctness in the pathway from high level languages to realistic low level implementation. ACL2 is the natural successor to Nqthm and has seen continued use, being adopted by industry for software verification. Goerigk [35] used ACL2 to prove partial correctness of a compiler. The main drawback of using ACL2 is that it only provides a first-order language, limiting its expressiveness. This allows the prover to make more general assumptions when considering proof automation. However it does make it more difficult to specify certain requirements.

PVS Much of the initial interest in compiler correctness came from related ESPRIT ProCoS projects, which in turn led to the Verifix project. The main focus of Verifix [36] was to develop generalised methods for compiler verification, using languages such as Common Lisp, ML and C as case studies. These efforts mostly used the Prototype Verification System (PVS). This proof tool is based on a classical typed higher order logic. It provides a collection of theorem proving procedures, which can be augmented by user defined procedures. As a result, the user can develop efficient automated elements specific to their needs.

LF, Elf and Twelf The LF logical framework is tailored towards the formalisation of various logics and programming language theory. Hannan and Pfenning [37], used an implementation called Elf to verify a compiler from Mini-ML, a simple functional language, to a simplified version of the Categorical Abstract Machine (CAM). The reasoning given for this choice of proof tool is the ability to use the same framework to specify all components, from the syntax and semantics to the compiling translation.

The latest implementation of LF is Twelf. Although it remains a lightweight solution for proofs in its specialised domain, it does not provide the automation

tools of other more general provers. As the general consensus is that some level of proof automation is invaluable for a system as complex as a realistic compiler, there are other technologies that are better able to provide this.

HOL and Isabelle The Higher Order Logic (HOL) series of theorem provers are successors to the LCF proof checker used by Milner and Weyhrauch [29]. Various instances of these are used throughout compiler verification projects. In particular, post-2000 papers provide a cluster of work using Isabelle, a popular interactive HOL theorem prover. These include Klein and Nipkow's [38] verified bytecode verifier for the JVM. Moving away from the specific area of compilers, Isabelle was also used for the sel4 verified microkernel project [39]. This was a significant result in the general field of system verification, and demonstrates that Isabelle is a potentially suitable tool for attacking these kinds of problems.

HOL4 is another theorem prover based on HOL. It formed a central part of the the CakeML[31] work towards a verified implementation of ML. This involved a verified compiler from a major subset of Standard ML to realistic machine code. It also looked at bootstrapping a correct compiler. CakeML showed that realistic compiler verification is possible for higher level languages. One of the main lessons from the project was the importance of constructing a compiler with verification in mind, such that the required invariants are preserved at each stage. The work also fed back into efforts to self-verify the HOL system itself [40].

Coq The Coq proof assistant has been a stalwart of verification work in recent years, including the landmark CompCert project [27]. Coq [41] is an interactive theorem prover written in OCaml, which implements a dependently typed functional language. The constructive logic used is based on the Calculus of Inductive Constructions. At the heart of the structure of Coq is the Curry-Howard Isomorphism, stating the equivalence of proofs and programs, and therefore of propositions and types.

Hence, Coq provides a type theory called Gallina which is used to specify

definitions, theorems and proofs, and is equipped with a corresponding type checker. A layer of syntactic sugar makes this more accessible to the user, and a tactic language called Ltac can be used to write proof scripts which generate Gallina proofs. The equivalence of proofs and programs also facilitates the Coq extraction mechanism. This allows users to develop specifications in Gallina and then extract the corresponding program in OCaml, Haskell or Scheme. Extraction to certain other languages is also possible via community developed plugins.

There are many features that make Coq a valuable proof tool. Coq is dependently typed, which means that types can contain references to the program itself. Therefore the user can write types which express correctness properties. This could potentially lead to more elegant proof techniques. In addition to this, Coq is based on a small kernel language which can be independently verified. This strengthens the user's ability to trust the proof tool itself. Furthermore,Coq provides avenues to a certain degree of proof automation. Alongside some basic automation tactics within the standard Ltac language, users can write their own tactics using Ltac or by plugging in ML code.

Across various works, Chlipala [42, 43, 44] argued for the importance of appropriate automation in compiler verification, choosing Coq as a suitable tool. With complex proofs,there are often many cases to consider of which only a handful will embody the crux of the proof. Hence where uninteresting cases can be made trivial by automation tactics, the user's time can be better spent on understanding the core nature of the problem. Moreover, where language specifications and theorem statements change, fully manual proofs can require a large amount of editing in order to extend to the new situation. Well written proof scripts, in a tactic language such as Ltac, can be structured such that they will adapt to changing specifications.

Chlipala [42] used Coq to certify a type preserving compiler from simply typed lambda calculus to an idealised assembly language. The paper claimed that the type preservation and use of typed intermediate representation facilitates efficient automation, through a type directed proof search. Where the

types are rich enough they can inform decisions as to which proof step to take, with sufficient accuracy to be useful. Hence, Chlipala demonstrated that the Coq mechanisms are suited to this kind of partially automated reasoning.

2.3.3 CompCert and program simulations.

CompCert [27] was a landmark result, not only demonstrating the feasibility of compiler verification but also setting out a template for achieving it. The CompCert compiler is an optimising compiler for Clight, a subset of C aimed at security critical applications. Leroy argued that it was easier to construct a compiler with the express intent of verification, rather than trying to verify an existing compiler. As such, CompCert is structured as a series of compiler passes, over a series of intermediate languages, all formalised using Coq.

In CompCert, the notion of correctness says that the compiler will either fail to compile, or produce target code whose behaviour is an acceptable behaviour of the source code. This is formalised and encoded as semantic preservation. Intermediate languages each have a small-step operational semantics, specified as a series of program transitions. Semantic preservation is then proved by simulation: each transition in the source language must have a corresponding series of target-level transitions, which have the same observable behaviours and preserve some relation between source and target execution states.

Subsequent work on compiler correctness has sometimes made use of this simulation technique. One such example is Lochbihler's compiler [45] for Java threads. Using the Isabelle/HOL proof assistant, rather than Coq, Lochbihler leveraged a similar idea of proof by simulation.

2.3.4 Compositional compiler correctness.

One problem with verification using program simulation is that it assumes whole-program simulation. In order to, essentially, run source and target code, the code segments must be complete and self-contained. However, in reality code is most often written and compiled as components which are then linked

together and even linked with external elements such as libraries. In which case, we lack complete programs at source and target levels to reason about.

Compositional compiler correctness looks at being able to verify the compilation of components and how they will behave when linked at target-level. Some research focuses on extending CompCert to this effect. SepCompCert [46] considered components that are compiled using the same compiler and then linked. Further to this, Compositional CompCert [47] aimed to reason about components compiled from different languages, but still from one of the CompCert family of intermediate languages.

Other work has looked more generally at what we mean by compositional correctness. Patterson and Ahmed [48] aimed to standardise the myriad of compositional theorems and proofs, in order to make the field more approachable.

There are typically two dimensions of compositional compiler correctness. Horizontal compositionality considers whether correct compilation guarantees are preserved when compiled program components are linked together. Vertical compositionality considers how correctness guarantees propagate through a compiler with multiple phases.

Rather than translating source code to target code all at once, most compilers perform a series of smaller rewrites. Each one of these rewrites makes a particular change, eliminating a high-level language feature or optimising the underlying program, advancing towards the target language. In complex software, like a compiler, such modularity allows for easier maintaining and debugging. It also makes the compiler easier to reason about.

Most verified compilers are also structured as a series of passes. For instance, CompCert consists of 14 passes over 8 intermediate languages. This necessitates a formalisation of each intermediate language. However, it also makes each compiler pass simpler and thus more straightforward to verify. Moreover, if the correctness property is appropriately chosen, each pass can be verified individually using the technique most suited to it.

In the majority of verified multi-pass compilers, the correctness property is chosen to be transitive. This means that if a translation from language A to

language B is correct and a translation from language B to language C is correct, then the result of composing these translations, to create a translation from A to C, will also be correct. Hence, it is possible to verify a multi-pass compiler by verifying each pass separately.

However, Patterson and Ahmed [48] claimed that transitivity should not be conflated with vertical compositionality. In many cases, although the correctness property may be transitive, verification of one pass still requires knowledge of the other passes or intermediate languages in the compiler. For example, Compositional CompCert relies on an interaction semantics encompassing all possible languages and memory transformations in the compiler pipeline. Therefore, if a new language or pass is introduced then the correctness of all other passes must be re-established. Patterson and Ahmed argued that true vertical compositionality should allow compiler passes to be verified entirely separately, with no regard for the rest of the compiler.

2.3.5 Types and compilers.

A significant amount of work on reasoning about compilers has focused on secure compilation [49]. This is a particular concern where source-level abstractions, relied upon to enforce some security properties, are not available at target-level. For example, if a source language is strongly typed, but its corresponding target language is not, then it is possible for a target-level attacker to violate the guarantees provided by well-typedness. Therefore, it can be advantageous for program transformations to preserve types.

For compilation to be type-preserving it must compile well-typed source code to well-typed target code. This necessitates a typed target language, along with typed intermediate languages for a multi-pass compiler. Thus, typechecking can be performed at each stage, ensuring type safety is preserved and aiding in debugging.

Although it is beneficial, type-preservation is not sufficient to imply secure compilation. A compiler targeting a unityped language could technically be

type-preserving, without actually enforcing any of the useful source-level security properties. It is therefore important to carefully examine the particular type systems and type translations involved.

Security-typed [50] programming languages implement types directly representing security properties, allowing the programmer to label data according to security levels. For a translation to be security-type-preserving both source and target type systems must satisfy non-interference, as well as the translation being type-preserving. Non-interference means that in a well-typed program low-security outputs depend only on low-security inputs, and will not be changed by changing high-security inputs. For secure compilation it is also necessary for the compiler to be correct, and that security levels assigned to target-level variables are the same as the corresponding source-level variables.

In addition to the security benefits, having typed intermediate and target languages can also help the compiler to make more effective optimisations. TIL [51] is a compiler for Standard ML which makes use of typed intermediate languages to perform type-directed optimisations. Such optimisations are deemed to have a beneficial performance impact for the resulting program, with respect to the cost incurred at compile-time.

In TIL types are only propagated up until a certain point in the compiler pipeline. Morrisett et al. [52] took this further, compiling System F all the way to a typed assembly language (TAL). This allows the enforcement of source language abstractions at the level of machine code, with programs in TAL essentially becoming proof carrying code. It also means that, before linking with and executing untrusted code at target-level, suitable checks can be made.

2.4 Modular Reasoning Techniques

The merits of formal methods with respect to testing have been much debated. Although it can be costly process, software verification provides proof that software will behave as expected for any possible input, as opposed to the generally limited cases considered by testing. Thus, verification results in more reliable

software, which carries with it a guarantee of this reliability. In addition to this, implementing verified software forces the developer to consider what behaviour is required of it. For instance, writing a postcondition for a program segment necessarily distils and formalises the intended behaviour of that segment.

Where software is modular, it makes sense for the verification techniques used on it to also be modular, thus exploiting the benefits, like clarity, robustness and reusability, to the fullest extent. When reasoning about software, there tend to be two different types of modularity to consider: parallel and sequential. Parallel modularity draws inspiration from the fields of high performance computing and concurrency, considering components that are running in parallel. Sequential modularity deals with control flowing sequentially from one component to the next, which more accurately reflects many modular systems.

Fisler and Krishnamurthi [53] developed a technique for modular verification of collaboration based software design, of which sequential modularity is an example. Here collaborations are modules which reflect the features implemented by software, rather than decomposing into modules which represent the actors which are used to implement such features. The paper argues that this is a more realistically useful sense of modularity. By combining parallel and sequential modularity, a feature-oriented approach to modular verification can closely mimic the modules used in software design [54].

2.4.1 Hoare-style postconditions.

The Hoare logic [55] framework was designed for reasoning about program correctness. Hoare triples still form the basis of specification in many commonly used verification techniques. A triple states that if a particular precondition is true, and a given program is run and terminates, then a given postcondition will be satisfied. Thus, Hoare triples can be used to express the expected behaviour of a program. The other key ingredient of Hoare logic is a modular approach to proofs, designed to echo the structure of modular software.

The Hoare-style postconditions are logical assertions, as are the precondi-

tions. In essence, such an assertion is a predicate, that is a function mapping the program state to Boolean values. This is often implemented as a proposition that captures the variables of the program in question. These assertions serve a wide variety of purposes in application, including: documenting assumptions and expectations, specifying theorems for static analysis to check, and forming conditions to test at runtime.

2.4.2 Design by contract.

Design by contract is a paradigm that involves inserting into the code itself specifications about the behaviour of the program. In essence, the contract makes a promise that must be upheld by the resulting program. Drawing on the earlier ideas of Hoare logic, contracts are made up of assertions, including preconditions, postconditions and invariants, which can be optionally checked at runtime to validate corresponding code properties.

First developed as part of the object oriented Eiffel language [56], design by contract is often associated with object oriented programming [57]. The aim was to outline a methodology for developing correct and robust software in the growingly popular object oriented style. The propensity for code reuse was viewed as a concern with respect to reliability. It means that any bugs are potentially multiplied across all of the times that a component is used, and that all possible conditions of use have to be considered when evaluating correctness.

While some languages, such as Eiffel and JML [58], are specifically designed to incorporate the use of contracts, other languages need to exploit libraries, such as Scala Predef [59], which mimic a suitable extension. By default, Scala imports four operations: assert, assume, require, and ensuring. Together these operations can be used to construct effective contracts within Scala code, which integrate smoothly with the way that the language handles modularity in general.

On a more formal level, contract theory is central to the area of assume-guarantee reasoning, developed to reason about program interference in cases

such as concurrency [60]. Assume-guarantee reasoning revolves around specifying triples which contain a component of the system, an assumption about that system, and a guarantee that will hold as long as the assumption holds. These triples can then be composed, to reflect how the system components are composed, and to clarify which assumptions must be upheld by the guarantee of any given component. This idea of safe interference allows modular reasoning about realistic systems, in which components are likely to share various resources.

2.4.3 Property based testing.

Rather than embedding specifications, some testing frameworks lift the required properties, such as postconditions, out of the code itself. This makes it easier to look at the potential interactions of multiple code segments, instead of reasoning about one point in the code at a time. QuickCheck [61] allows programmers to implement property based testing within Haskell code. Given a postcondition, for example, it can automatically generate correspondingly appropriate tests and execute these on random appropriate inputs. Similar tools for other languages have been inspired by this approach, such as ScalaCheck.

Property based testing can exploit ideas from design by contract more directly. For instance, Cheon and Leavens [62] developed an approach to unit testing using JUnit and JML together to generate useful test cases. JUnit is a unit testing framework for Java classes which automates some of the details, thus streamlining the process. JML, or the Java Modelling Language, is a behavioural specification language that facilitates the writing of contracts for Java. Together these tools can be used to automate the implementation of test oracles, using the contract specifications and assertion checker of JML to determine whether unit tests pass or fail. This not only reduces the developer effort required, but the written specifications are generally clearer and more compact than extensive test code.

2.4.4 Automated theorem proving.

Automated verification techniques are often used as a realistic middle ground between the relative accessibility of testing and the expressiveness of theorems proved by more involved means.

Static code analysis tools can also make use of contract-like syntax. For instance, Leon [63] is a verification tool working over Pure Scala, a functional subset of the Scala language. It uses Scala contract assertions, like require and ensuring, to generate corresponding verification conditions, which are handled automatically by SMT solvers such as Z3 [64] and CVC4 [65]. If a program meets its specification then Leon will seek to prove this; otherwise it will seek to find a counterexample.

Finite state machines can be used as a mechanism for specifying the behaviour of software. Model checking tools can then automatically check programs against a corresponding state machine. For example, the MAGIC tool [66] for modular analysis of programs in C, can be used to verify that a given state machine is a safe simulation of the C procedure that it relates to. The tool adheres to modularity, in that it can accept and act on specifications for missing parts of the system. The modular approach ensures that the components being verified are of a manageable size, avoiding the danger of state-space explosion, which is a sticking point for most automated verification efforts.

2.4.5 Interactive theorem proving.

The field of proof engineering revolves around developing verification approaches that scale to realistic software engineering applications. A significant amount of this work involves the use of proof assistants, otherwise known as interactive theorem provers [67]. Proof assistants, such as Coq [41] and Isabelle [68], facilitate the rigorous development and checking of proofs via a combination of human input and machine reasoning.

Despite the increase in human effort required, interactive methods such as the use of proof assistants have various benefits over more automated tools. It is

often possible to develop proofs of properties interactively that would otherwise be undecidable. Proof assistants also usually depend on a small kernel to check proofs, according to the de Bruijn criterion, thus reducing the potential impact of any bugs in the tool itself.

Another verification approach is to combine program code with the use of a proof assistant. There has been work [69] on translating Pure Scala into Isabelles/HOL logic, with Isabelle used as a backend for Leon. Moreover, Kopitiam [70] is an Eclipse plugin that allows development of verified Java code by interfacing with Coq. Methods are annotated with specifications, which are verified by the developer using Coq and otherwise ignored by Eclipse.

Scala is both object oriented and functional, but the Pure Scala fragment, typically used in Scala verification efforts, focuses on the functional elements. Properties of purely functional languages, such as referential transparency, tend to make verification less complicated. In contrast to this, Kopitiam leverages intuitionistic higher-order separation logic, in order to effectively reason about the object oriented features of Java.

2.5 Summary

This chapter took a broad look at the areas of compiler structure, compiler verification, and general correctness approaches, examined through the lens of modularity. The selection of background material that we explored, supports the significance of three key compiler qualities: *modularity*, *performance*, and *correctness*. Modular compilers are easier to understand, maintain and debug. Performant compilers are inevitably more popular and contribute to user productivity. And correct compilers are vital for avoiding awkward software bugs.

However, we began to see that trying to achieve all of these qualities simultaneously results in a tricky balancing act. For instance, improving modularity can increase the number of compiler phases, which in turn can negatively affect compiler performance. Settings like the miniphase framework [1] seek to address this via automated fusion techniques. Such approaches maintain that

phases can be modular from the perspective of the compiler developer, in a way that avoids actually having large numbers of phases at runtime, and without added effort for that developer. Standardised compiler phases can be automatically combined in a standardised manner, thus mitigating the overheads of performing them individually.

Chapter 3

Related Work

Once compiler modularity and performance have been attained to an acceptable degree, correctness is the remaining concern. Ensuring the correctness of compiler phases can be an expensive process, and so it is unproductive for any automated fusion mechanism to endanger this. The miniphase framework uses postconditions to allow developers to specify the invariants established by compiler phases. These postconditions are then checked during the testing process of the final fused compiler, an approach which is practical but not particularly formal or modular.

This chapter narrows the literature focus, to look at ways in which existing work tries to achieve modularity at the same time as performance and correctness. Rather than thinking about compiler phases in general, we specifically target approaches that explicitly concern graph transformations, in an effort to find work that is directly comparable to the tree transformation based miniphase framework. Furthermore, this chapter explores tree transformations for applications outside of compiler construction, to look for techniques that could also be useful for compilers.

3.1 Miniphase Fusion

Petrashko et al. [1] proposed and implemented miniphase fusion in the Dotty compiler for Scala. Dotty was originally a platform for experimenting with ideas for the Scala language, and was intended to form the basis for Scala 3.0. Historically, a frequent Scala complaint had been the length of compile-time. Hence, a major goal was to make the Dotty compiler faster, whilst also imposing a more modular approach. The modularity was approached through the introduction of miniphases, which were billed as *"modular and efficient tree transformations"*.

3.1.1 Motivation for miniphase fusion.

The most common technique for compiling Scala to Java bytecode involves a series of abstract syntax tree (AST) transformations. An AST, produced from the Scala source code, is gradually transformed to be closer to Java bytecode, through a process of desugaring and rewriting into lower-level language features. Ideally these tree transformations would be modular, each performing a small singular task within its own tree traversal. Modularity is important in the readability and maintainability of complex systems, such as a compiler.

However, modularity tends to encourage a large number of AST transformations, which can have a negative impact on performance, due to a correspondingly increased number of tree traversals. Since the ASTs used in compiling Scala are generally larger than cache size, these traversals can be very expensive. Moreover, the tree nodes are immutable and have no link to their parents, so trees must be entirely reconstructed each time even a small change is made. Therefore, a strategy for mitigating these performance issues is needed if a modular approach is to be viable.

3.1.2 The miniphase fusion mechanism.

Previous Scala compilers tended to prioritise performance over modularity, by manually combining multiple traversals into the same traversal. The burden and implementation of such decisions is placed solely on the compiler developer. The result of this is compiler phases which may perform many unrelated tasks, and so are difficult to understand. These phases are also hard to reason about, as the vastly different transformations may interact in unexpected ways.

The miniphase approach attempts to standardise and automate the combination of transformations, mitigating the performance impact whilst maintaining a high level of modularity for the compiler developer. A miniphase class is implemented which must be extended by all compiler phases, thus imposing a common structure on them. A miniphase has: a separate transformation method for each type of tree node, a `transform` method which dispatches one of these transformations as appropriate for a given node, and a `runPhase` method which performs a postorder tree traversal, calling `transform` at each node.

The standardised postorder traversal is key in allowing miniphases to be automatically fused together. On a single postorder traversal, each node visited can be transformed by multiple miniphases, before moving on to the next node. To this end, two miniphases can be combined to make a new miniphase. A new transformation method is constructed for each node type which applies the corresponding transformation from the first miniphase and then, checking the type of the new resulting node, applies the corresponding transformation from the second miniphase.

There are several extensions to this basic miniphase idea, that make it more suited to the Dotty compiler. The fusion process makes certain optimising decisions, such as skipping identity transformations and exploiting transformations which do not change the type of a node. There is also the addition of `prepare` methods, making it possible to implement a groups of transformations which would otherwise require a non-postorder traversal, due to depending on a node's ancestors.

Petrashko et al. present both experimental and anecdotal evidence for the success of miniphases in Dotty. In terms of performance, when compiling the Scala standard library and the Dotty compiler itself, miniphase fusion decreases the time taken on tree transformations by 37% and 34% respectively, compared to the same unfused miniphases. This is deemed to be due to more effective CPU cache use, with a reduction of 49% and 55% respectively in the objects promoted to old generation.

3.1.3 Reasoning about miniphase fusion.

A key concern in fusing tree transformations is how it changes their behaviour. Once fused, multiple transformations are performed on a single traversal, as the transformations are interleaved. Hence, inherent assumptions about the whole tree, which may hold if a single transformation is performed alone, may be violated by other transformations working on the tree during the same traversal. Such interactions between transformations have the potential to produce unintended or incorrect behaviour.

If fusing miniphases stops them from performing the tasks they were designed for, then fusion is no longer useful, regardless of how modular or efficient the outcome may be. To address this, Petrashko et al. outline a set of guidelines for developers, on when miniphases may be successfully fused. A miniphase may be fused into a block of already fused miniphases if it achieves the the following.

- preserves the invariants established by all prior miniphases in that fused block
- successfully transforms tree nodes with children that have been transformed by all subsequent phases in that fused block
- successfully transforms nodes in a tree that has not yet been completely transformed by all prior phases in that fused block

These are described as *"high-level criteria"* to be *"interpreted with an understanding of the overall design of the compiler"*. The argument against more

formal criteria is that they would have to be overly conservative and thus inappropriate for capturing the complexities of a realistic production compiler.

In addition to these criteria, successful fusion is ensured by a series of dynamic checks during testing. Each miniphase has a postcondition check, which can be implemented to enforce some property of an AST that must hold after being transformed by that miniphase. The idea is that all subsequent miniphases must preserve this postcondition, once it has been established. During testing, an extra pass is inserted after each phase, to check the postconditions of all miniphases that have run so far.

Dotty miniphases also have the notion of preconditions, expressed in terms of the postconditions of other miniphases. These preconditions take the form of lists, `runsAfter` and `runsAfterGroupsOf`, denoting miniphases that must have previously transformed a node or previously transformed the entire tree, respectively. This information is used by compiler developers in ordering miniphases and splitting them into different fused blocks, and is checked when the compiler first starts up.

The high-level guidelines, the postcondition checks, and the lists of preceding miniphases must all be taken into account when determining how the compiler is structured. Petrashko et al. present a pipeline for the Dotty compiler that begins with a series of unfused frontend phases, including parsing, typechecking, and some transformations to ensure the AST is in the format expected by the rest of the compiler. This is followed by six blocks constructed of fused miniphases, and ends with a phase to generate target code, such as Java bytecode.

The miniphases, that comprise the middle section of this compiler, cannot be all fused into one block due to various conflicts that are alluded to by the high-level criteria for successful fusion. For example, some miniphases make wide-reaching changes to the structure of the AST, as they translate very high-level features into lower-level constructs. One such miniphase translates Scala pattern matching expressions into a complex series of branches and jumps. Before this miniphase has entirely transformed the AST, the tree will contain parts of both

the original pattern matching and its translation. It can then be difficult for some other miniphases to handle both high-level and low-level constructs in the AST at the same time, and thus they should not be performed on the same traversal as the pattern matching miniphase. In this way, miniphases that make global AST changes can necessitate the separation into multiple fused blocks.

3.1.4 Comparisons with the nanopass framework.

Along with miniphases, the Background chapter introduced the nanopass framework [11, 12], as an alternative approach for implementing structurally modular compilers. Nanopasses also impose a given structure on compiler phases, which allows the common functionalities to be abstracted out, and thus promotes code reuse. Furthermore, the nanopass has also been successfully applied to a compiler in a commercial setting, validating the approach. Thus, there are many similarities between the aims and accomplishments of the miniphase and nanopass frameworks, with respect to modularity.

However, the two approaches diverge somewhat when considering the issues of performance and correctness. The original nanopass framework [11] was aimed at students learning how to implement a compiler. As such, it makes sense that production level performance is not a central concern. There is no mechanism intrinsic to the nanopass framework for directly addressing the trade-off between modularity and performance. However, the paper does discuss as future work the idea of a phase combiner based on deforestation techniques, which is similar to what miniphase fusion later achieved. The advantage that miniphases have is that their whole design revolves around the fusion process, rather than implementing it as an afterthought.

In terms of correctness, where miniphases have postconditions, nanopasses have formally specified intermediate languages. These elements serve loosely the same purpose as each other, in that they encode guarantees that need to hold, after a given compiler phase, about the program being compiled. Each nanopass has to specify an input language and an output language, as well as

transformations from input language features to appropriate target language features. These intermediate languages are well suited to encoding properties like the absence of a given language feature. However, tying a compiler phase to a specific point in the compiler pipeline, reduces the opportunities for direct reuse or easy rearrangements. On the other hand, the miniphase approach uses the same AST representation throughout the series of tree transformations, which results in no such restrictions.

3.2 Fusing Tree Traversals

There is a significant body of relevant existing work that focuses on fusing tree traversals. This provides an appropriate point of reference for assessing the merits and limitations of the tree based miniphase approach. We make an important distinction here between the connotations of the terms tree traversal and tree transformation. A tree traversal generally refers to visiting the nodes of a tree for the purpose of calculating some value. Changes may be made to the nodes visited, but tend to be very restricted, making it easier to provide rigorous static correctness guarantees. A tree transformation, on the other hand, takes a tree as input and outputs another tree, being not so limited in the changes it may make.

3.2.1 Exploiting temporal locality.

Miniphase fusion was deemed to be successful in improving performance due to improved cache locality. There are tree traversal fusion techniques that seek similarly to exploit temporal locality. By running close together the traversals which visit the same nodes, the node data will still be in cache, making it less costly to retrieve.

Given a series of points that traverse and interact with a tree, point blocking [71] involves sorting these points into blocks, depending on the nodes that they visit. Each block performs a single tree traversal and, at each node, all points

in the block that interact with that node are applied. Point blocking relies heavily on preprocessing to sort the points into appropriate blocks. Traversal splicing [72] is a similar technique that sorts points dynamically, as they are being applied, to enhance locality.

3.2.2 Exploiting static dependence analysis.

The original work [71, 72] on point blocking and traversal splicing focused mainly on independent tree traversals. Weijiang et al. [73] developed a static dependence test to extend these techniques to a wider range of traversals, which may interact with each other. In analysing the node access path in the traversal algorithms, the test determines whether point blocking or traversal splicing could be applied safely and when node visits can be reordered.

Rajbhandari et al. [74, 75] looked at automatically finding the optimal fusion schedule for recursive traversals of k-d trees. In particular, they considered the MADNESS system, which is designed for numerical scientific simulations. They examined the data dependences of traversals based on their consumer-producer relationships and showed that fusing operators by interleaving them can improve performance by improving locality, as the trees used are often larger than the cache size.

3.2.3 Exploiting code motion.

TreeFuser [76] is a framework that looks to automatically fuse more general tree traversals. It employs code motion and partial fusion to perform as much fusion as possible. Code motion involves rearranging code such that fusion becomes feasible, for instance by changing the traversal order. Partial fusion considers traversals that cannot be fused completely, but can be fused over parts of the tree, still improving performance. TreeFuser produces a dependence graph that is used to determine when these techniques are applicable.

3.2.4 Exploiting logical reasoning frameworks.

Qiu and Wang [77] implemented a decidable fragment of the DRYAD logic for reasoning about trees. DRYADdec is especially suited to analysing tree traversals which calculate some measurement of the tree. One sample DRYADdec application presented is to check whether the fusion of a certain set of tree traversals is allowed, that is whether the fused traversals will have identical behaviour.

3.2.5 Comparisons with miniphase fusion.

All of this work on fusing tree traversals takes a fundamentally different approach than that of miniphase fusion. Firstly, the tree traversals can only make minor changes to the nodes being visited. It may be possible to rewrite the data contained at a node being visited, but it is not possible to dramatically alter the structure of the tree by changing the children of that node. This is perfectly fine for some applications. However, AST transformations in a compiler such as Dotty are required to make more drastic changes.

In addition to this, many authors are concerned with rearranging tree traversals to yield optimal fusion. For instance, point blocking and traversal splicing purposely sort the tree traversals in order to group them. On the other hand, miniphases in Dotty are generally written with a specific order in mind, partially informed by the preconditions stated. This is all the more significant being a series of tree transformations in which the output of each transformation becomes the input for the next.

Finally, the definition of successful fusion in these approaches is that the behaviour of fused traversals should be identical to that of running them consecutively. In TreeFuser fusion must preserve work, that is, exactly the same set of operations must be performed whether fused or not. This makes sense for traversals which are, for example, calculating a value from the tree. However, in compiler phases the goal is often not deterministic, for example eliminating a given syntactic element from the tree. It is possible to preserve this goal without

requiring fusion to exactly preserve the results.

3.3 Deforestation and Stream Fusion

The miniphase framework claims deforestation as its inspiration. It is common to find, especially in functional programming, a series of functions applied successively to a data structure, such as a list or a tree. The intermediate data structures generated, as the output of one function and input of the next, can have a performance impact. Deforestation [78] automatically combines such sequences of functions so that intermediate data structures are not required. This allows developers to write complex functions by composing simpler ones, without the cost to performance effected by intermediate lists or trees. The Glasgow Haskell Compiler (GHC), for instance, performs a similar optimisation on code during compilation.

3.3.1 The origins of deforestation.

As originally proposed by Wadler [78], deforestation deals with fusing a very restrictive set of functions. In order to be fused, the composed functions must be in treeless form, which means they must be first-order, use all variables linearly, and not construct any internal data structures themselves. These restrictions make it possible to prove that the deforestation algorithm will terminate and that the result will be treeless.

A number of efforts have been made to extend deforestation to a wider range of functions. Often this involves identifying functions which cannot be fused and abstracting them out of the process, as with Wadler's blazed deforestation [78] which caters to functions whose subterms may have atomic types such as integer or char.

3.3.2 Safety definitions in deforestation.

Chin [79] generalised deforestation to all first-order and higher-order functions. Functions are determined as safe or unsafe to fuse depending on how they produce and consume data structures. An important distinction is made between functions that can be safely fused, meaning the algorithm will terminate and there will be no loss of efficiency, and functions that can be effectively fused, meaning that intermediate data structures will be eliminated and performance will be improved.

Similarly, shortcut deforestation [80] for Haskell takes all legal programs as inputs, guaranteeing that the process will terminate but not that all intermediate data structures will be removed. This is done by standardising the way that lists are produced and consumed, using foldr/build pairs which can be cancelled. Shortcut deforestation was incorporated into the Haskell List library in the GHC. This has since evolved further, and the GHC now uses stream fusion [81, 82] which applies to a wider range of intermediate data structures.

3.3.3 Deforestation and category theory.

Many deforestation and fusion frameworks do not provide formal correctness guarantees, relying instead on implementation and testing. They often use a syntactic approach which makes it easy to describe the implementation of fusion, but difficult to construct appropriate proofs. However, some cases use category theory to look at the semantics of fusion, which can facilitate the proof process and generalisations.

Takano and Meijer [83] used category theory to extend shortcut deforestation [80] beyond lists, to other data structures. Generalising the idea of foldr/build pairs, a hylomorphism is made up of an anamorphism followed by a catamorphism. Like build, the anamorphism produces a set of results and then, like foldr, the catamorphism combines these results into a single value. Existing results on the fusion of hylomorphisms [84] can then be applied to the case of deforestation.

3.4 Equality Saturation and PEGs

The equality saturation [23, 85, 86] approach to compiler optimisations discards the traditional sequential method of stringing transformations together. This technique focuses on the optimisation section of the compiler pipeline, aiming to tackle the phase ordering problem. In a conventional compiler structure, the order in which optimisations are applied will have an impact on the quality of the eventual target code. It may be that one optimisation acts to limit the effectiveness of subsequent optimisations. Typically, there is no single ordering that will always ensure the best optimised result for all inputs. Moreover, testing all possible orderings of modular optimisations can be prohibitively costly, especially in cases where optimisations are designed to be repeatedly applied, until there is no further change.

Equality saturation avoids this problem by taking an additive approach rather than a destructive one. The traditional sequencing of optimisations means that once any given optimisation has been applied, the original unoptimised program is lost. Only the optimised version is passed on to the next optimisation. In equality saturation nothing is lost. Instead, each optimisation simply adds information, in the form of axioms denoting possible rewrites, without having to prematurely decide on an order. It is then left to an appropriate global profitability heuristic to analyse these annotations and determine the best result, once all information is known.

The structures used for Tate et al.'s [23] presentation of equality saturation are based on program expression graphs (PEGs). The PEG nodes correspond to operators, including inbuilt operators that denote conditionals and loops. To facilitate equality saturation, the E-PEG is a graph that groups PEG nodes into equivalence classes, thus representing multiple possible optimised versions of a program at the same time. Essentially, an E-PEG is a PEG augmented with a set of equalities between nodes.

This kind of program representation allows local graph rewrites that make significant changes in structure of the respective CFG. Therefore, it is a good

vehicle for reasoning about realistically useful optimisations, such as loop optimisations. PEGs are referentially transparent, which means that there need be no consideration of side effects when trying to determine equality. Only the value of a program fragment is significant, and this can be represented by a single node, rather than having to examine an entire subgraph. Therefore, equivalence classes of nodes are sufficient for encoding the equivalence of parts of programs, for use in equality saturation.

3.5 Conclusions: Background and Related Work

Fusing transformations is beneficial in terms of both modularity and performance.

Modularity, by its nature, necessitates joining components together. The simplest ways of doing so are not always the most efficient, due to elements that make sense for an individual component but not when combined with other components. Deforestation automatically combines functions to avoid creating unnecessary intermediate data structures, allowing the developer to concentrate on modularity with the reassurance of mitigated performance impact.

Building on the idea of deforestation, miniphases in the Dotty compiler allow automatic fusion of tree transformations. Here the intermediate trees are necessary, as transformations need to consume and produce subtrees. However, there is also the issue of modularity effecting an increased number of tree traversals, due to an increased number of transformations, each requiring its own traversal. Instead of applying these transformations consecutively, miniphase fusion combines multiple transformations into a single traversal.

These methods are used to good effect in practice. Stream fusion, which evolved from deforestation, is used by the GHC to rewrite Haskell code more efficiently. Moreover, miniphase fusion in Dotty was shown to reduce the time taken by tree transformations in compiling code, whilst providing a beneficial user experience for compiler developers. Therefore, automatically fusing tree

transformations is a valuable technique.

The standard view of successful fusion exactly preserves outcome, but this can be overly restrictive.

In most cases, the notion of successful fusion is defined as ensuring that, whether fused or run consecutively, the outcome of a series of transformations will be identical. This is a valuable guarantee, meaning that the developer and end user should see no effect of fusion other than, hopefully, an improvement in performance.

However, we find in some cases that this requirement is overly restrictive. For optimisations, fusion may alter the final outcome, and hence make the optimisations more effective. To achieve this otherwise would require both optimisations to be performed in a single transformation, thus reducing modularity. Therefore, we want to consider a broader idea of successful fusion that encompasses cases such as these, so that fusion may be applied as widely as possible.

Postconditions are common tools for expressing and reasoning about program requirements.

Miniphases in Dotty do not come with such a formal soundness guarantee. Instead, they combine a high-level set of guidelines, for the compiler developer to follow, with postcondition checks for each miniphase. These postconditions can be used to specify the required outcome of a miniphase, and are used to ensure that these outcomes are met, even when fusion is performed.

Although, in the case of Dotty, postconditions are checked dynamically during testing, postconditions can also be useful for static approaches to correctness. There are various approaches to verifying postconditions, including using automated static analysis tools and interactive proof assistants. Hence, the postcondition is a flexible and approachable mechanism for encoding developer intentions.

Chapter 4

Methodology

The purpose of this thesis is to develop and evaluate a technique for reasoning, in an appropriately modular way, about combining modular compiler phases. Over the previous chapters, we have seen that existing work tends to compromise, either on modularity of reasoning or on functionality of program transformations. We explore the possibility of achieving a useful level of both.

From the literature, we have established that performance, modularity and correctness are all valuable compiler properties. Automatically combining compiler phases can facilitate modularity and whilst mitigating performance loss. However existing approaches to correctness in such cases tend to have low levels of modularity themselves. Hence, this thesis proposes modular criteria for successful fusion, which can be verified on separate uncombined compiler phases.

This chapter outlines and justifies the approaches taken in the remainder of this thesis. The first half of this chapter discusses the research questions being addressed, setting out the scope of the problem and some of the challenges. The latter half of this chapter details the corresponding research strategy, including the tools and techniques used.

4.1 Research Problem

4.1.1 What does it mean for compiler phases to be fusible?

The purpose of automatically combining compiler phases is to promote both modularity and efficiency within the compiler. However, it is vital that combined phases still work as intended. Compiler correctness is highly valued and hard won, generally requiring substantial effort to confirm. Therefore, we cannot afford to overlook correctness in the search for other desirable compiler qualities. A compiler that is elegantly implemented and quick to compile but does not produce the intended target code is ultimately of limited use.

The related work that we have examined has mostly taken one of two approaches. Most tree transformation fusion strategies take the conservative choice of only allowing fusion if the results of fused transformations are identical to the results without fusion. On the other hand, the miniphase approach uses manually defined postconditions to express what behaviours of each transformation must be preserved. This allows, for instance, the fusion of optimisations for which the fused transformation produces a different but better-optimised result than the unfused transformations.

If combining compiler phases is generally beneficial, then it makes sense to seek to apply fusion as widely as possible. This echoes the approach taken by miniphases, leaving the compiler developer to decide and specify how much of their phase's behaviour is necessary and must necessarily be preserved by fusion. However, it is also important to balance practicality with rigour when considering correctness issues. The miniphase postconditions are used in testing, which can only provide limited guarantees due to the multitude of corner cases inherent in complex software such as compilers. This is where other, more restrictive, fusion criteria can be useful, providing strict soundness guarantees due to the strict guarantees they place on their fusible transformations.

Taking these factors into account, we formalise postcondition-preservation as a notion of fusibility, which provides strong guarantees without unnecessarily

preventing beneficial fusion. We take inspiration for this from the miniphase [1] postcondition approach. To formalise these ideas we define a system of tree rewrite functions, with corresponding postcondition predicates on trees that encode the desired outcome of each transformation.

4.1.2 Can we determine whether phases are fusible without attempting to combine them?

The purpose of automatically combining compiler phases is to promote both modularity and efficiency within the compiler. A degree of modularity benefits complex systems in various ways, such as ease of maintenance and code reuse. This is one of the fundamental motivating forces behind fusing phases. Therefore, it makes sense for the fusion process and the methods of reasoning about that process to also be inherently modular.

Ideally, we want to be able to assess compiler phases for fusibility without having to actually fuse them. To fully embrace and exploit modularity, there should be some level of separation between the implementation and the interface of each compiler phase. This allows a tree transformation to be re-implemented with minimal disruption to the other transformations in the pipeline. The surrounding literature establishes this as a key part of modularity [6].

The more formal, restrictive approaches to fusing tree traversals are able to provide criteria for fusibility that are entirely independent of the other transformations involved. As long as the transformations are sufficiently limited, for instance only able to alter the value at the current node, then any fusion is allowed and will have no effect on the result of the transformations. However, it may be that our definition of fusibility is more lenient and our transformations need to make wider changes on order to be useful.

Conversely, the miniphase approach requires phases to be fused in order to test whether stipulated postconditions will be preserved. There may be some developer intuition at the level of individual phases as to whether they can be successfully fused, but this is far from guaranteed. So, if a phase needs to be

altered, even slightly, the whole combination will need to be run and tested again.

Therefore, once we have decided on a definition of fusibility, we are also looking for a method of confirming this. This method needs to subscribe to modularity as far as practicable, in order to maintain a consistent approach. However, it also needs to fit our compiler use-cases to be of any notable benefit.

4.1.3 The scope of the problem.

Since this research problem is complex and nuanced, it is necessary to clearly define the boundaries of our work. Here we describe some interesting areas that are beyond the scope of this thesis, in order to better specify the problems that *are* being explored in this research.

Firstly, the order in which compiler phases are applied can have a major impact on the compilation process. In some cases, the correctness of a compiler phase depends heavily on a previous phase having been performed. In other cases, such as certain optimisations, the order might impact effectiveness, rather than correctness. Often, the most effective order of optimisations depends on the code being compiled. The problem of phase ordering is orthogonal to the aim of this thesis. As such, we assume that the compiler developer has already decided on an ordering, and we make no attempt to reorder the transformations.

Secondly, it can be challenging for a developer to fully and accurately specify the purpose of a compiler phase using postconditions. The postconditions considered in this research are defined as predicates over graph structures, which are checked recursively. The examples of postconditions used to illustrate our results use pattern matching to allow or disallow certain graph patterns. In this way we can express concepts such as the lack of a particular language feature, or the termination of a given optimisation. However, further work could investigate the range of postconditions for realistic compiler phases that can be expressed within this framework.

Finally, the work in this thesis does not permit for preconditions to be ex-

pressed by the compiler developer, when specifying transformations and post-conditions. This is similar to the approach taken in the miniphases of Petrashko et al. [1], whereby the only form of precondition is to specify which other miniphases must have already been run. Further research could explore the introduction of formally defined preconditions, and the complex logic that arises with respect to preconditions when transformations are fused.

4.1.4 Illustrative Examples

This thesis focuses on compiler phases based on graph transformations. In particular, we look at transformations on ASTs and PEGs, predominantly inspired by Dotty miniphases [1] and work on equality saturation [23] respectively. Therefore, the selected examples are inspired by these sources. We choose transformations that are realistic but simple, to illustrate the situation in a way that is clear but still applicable to the real world.

Generally, compiler phases on an intermediate representation fall into one of three categories: analyses of properties about the program, translations of features in one language to features in another, and optimisations to improve the performance of target code.

Analyses

We do not explicitly look at analysis passes for our illustrative examples. This is because we are particularly interested in transformations that make structural changes to the program representation, such as removing nodes from an AST. In addition to this, the analysis passes generally fit the restrictions of existing tree traversal frameworks, which already provide very strong soundness guarantees. However, if a given analysis can be performed by traversing and annotating the tree or graph nodes, then it can be considered as a minimal transformation, and so our definitions and criteria do still apply.

Translations

At a simplistic level, the main job of a compiler is to sensibly translate code from one language to code in another. Ideally, these languages are chosen such that they relate to each other in as straightforward a way as possible. However, in reality there may be many differences between the two, including different language features. Therefore, it is important to select an example that translates one language feature into another equivalent form, in order to assess the applicability of our work in this necessary case.

Optimisations

Arithmetic Constant Folding Constant folding is a straightforward but useful compiler optimisation, wherein constant expressions which can be statically evaluated are replaced by the result of this evaluation. This is a good illustrative example of a compiler phase, since we only require a simple language subset, such as arithmetic expressions, to demonstrate it. Moreover, the majority of compilers implement some kind of constant folding as, for little effort, it can have a significant impact on the effectiveness of more complex optimisations. We use arithmetic constant folding as a running example, starting in Section 5.2.2, Example 3.

If Expressions In a similar vein, we can optimise if expressions by removing program branches that will never be visited. For instance, if the condition of an if-then-else expression is statically known to be true or false, then it can be replaced by the corresponding branch. This is another fairly simple optimisation that can have a major impact on the size of the intermediate representation, and hence the emitted target code. It can also interact with constant folding on Boolean expressions in cases that the condition can be optimised to just true or false. Constant folding on if expressions is used as a running example, starting in Section 5.2.3, Example 4.

Loop Optimisations Loops are a common source of optimisation opportunities, because the code within them may be repeated, which amplifies any performance improvements made. Moreover, there can be significant overheads involved in executing the loop itself. As such, the world of loop optimisations provides a wide selection from which to choose an example. Since, the encoding of loops is one of the reasons that we extend our work from trees to graph representations, it make sense to look at an illustrative loop optimisation. In particular, we look at loop induction variable strength reduction (LIVSR) in Section 6.4.2.

A simple source language

As a source language to illustrate these transformations with we use a version of SIMPLE, a toy programming language used in work on PEGs and equality saturation [23]. This provides just a few useful program constructs, and avoids obscuring the examples with unnecessary elements. The grammar of SIMPLE is as follows.

$$s ::= s_1; s_2 \mid x := e \mid \texttt{if}\ (e)\ \texttt{then}\ \{s_1\}\ \texttt{else}\ \{s_2\} \mid \texttt{while}\ (e)\ \{s\}$$

$$e ::= n \mid b \mid x \mid op\ (e_1, \ldots, e_m)$$

$$b ::= \texttt{true} \mid \texttt{false}$$

$$n ::= 0 \mid 1 \mid 2 \mid \ \ldots$$

We use subsets of this language throughout our examples, for both trees and graphs, in order to highlight the similarities and differences between the two kinds of representation.

4.1.5 The Coq Proof Assistant

Any research that relies heavily on specifying theorems and then proving them needs to communicate clearly the validity of those proofs. There are some fields in which endless pages of mathematics and symbolic reasoning are welcomed, if

not expected. However, the very topic of this research seeks to straddle the line between formalisation and accessibility to a wider audience. We do not simply wish to derive and verify a set of criteria; we want those criteria to be useful and rational for practical compiler developers, whose expertise may not lie in formal methods. Therefore, it is important that a wide variety of reader could convince themselves of the veracity of the proofs in this thesis.

One major advantage of proof assistants, such as Coq, is the assurance that proofs have been mechanically checked. The reader is left only to convince themselves that the theorem in question is accurately expressive. This is often a less involved task, requiring more knowledge of the use-case than the formal framework. Moreover, the Coq Proof Assistant is known to be reliable. Hence, we use Coq to check and mechanise some of the informal proofs in this thesis. A stable version of these proofs can be found at:

`https://github.com/EleanorRD/PhDThesis` [87].

A significant point of clarification here is that the work in this thesis is not specific to Coq. The Coq proof assistant was chosen as a tool, for reasons including reliability and successful use in similar projects, but there are other proof assistants and other verification tools that could have been used to similar effect. However, this choice does promote a somewhat functional and recursive approach to specification and proof, which also influences the presentation of this thesis.

4.2 The Remaining Chapters

This chapter has set out the goals for the rest of the thesis, in the context of wider research questions and with some discussion of how we approach them. The remaining chapters explore these questions for compiler phases in the form of, initially, tree transformations and, then, more general graph transformations.

Chapter 5 begins by looking at fusing AST transformations, building on the existing miniphase approach. We motivate and define a concept of fusibility, in

terms of compiler phases with postconditions, and derive modular criteria for reasoning about this fusibility at the level of individual phases. Then we evaluate these criteria by proving their soundness and considering several illustrative examples of compiler phases.

Chapter 6 continues the same approach and looks to extend it to a more general graph structure. We examine why it can be useful to include shared nodes and loops in a program representation. We then look at an inductive approach to defining graphs, that naturally extends the inductive tree definition of the previous chapter. This choice of inductive graphs is what concerns the latter part of the chapter, discussing the applicability of this definition to realistic transformations.

Finally, Chapter 7 rounds off the discussion of this work, by means of comparison with appropriate related work. It also outlines some potential future work and sets out the conclusions of our research.

Chapter 5

Fusing Tree Transformations

Consider a series of abstract syntax tree (AST) transformations in a compiler. An initial AST is created by parsing source code. Then, this AST is passed through a series of transformations, the output of each transformation becoming the input of the next. These transformations act to remove high-level language features or to optimise the underlying program. Finally, target code is generated from the transformed AST.

One approach to using AST transformations effectively is to automatically fuse them, maintaining modularity for the compiler developer, whilst reducing the number of AST traversals required when the compiler is run. To this end, Petrashko et al. [1] proposed and implemented miniphase fusion for the Dotty Scala compiler. Miniphases impose a structure on AST transforming compiler phases that allows them to be automatically fused.

Miniphases rely on informal guidelines to advise the compiler developer as to when transformations may be fused. In most work on fusing tree traversals, soundness is formally proved, yielding stronger correctness guarantees [71, 72, 74, 75, 76, 77]. However, there are two factors that prevent the solutions

proposed in such related work being directly useful in the case of miniphase-style fusion.

Firstly, the traversals being considered tend to involve limited transformations. In general, changes cannot be made to the children of a node, only to the data stored at the node itself. This is overly restrictive for compiler phases which need to make drastic structural AST changes.

Secondly, successful fusion is usually defined as producing a fused transformation that will always give the same result as running its individual constituent transformations successively. However, we can demonstrate, with an example, that this precludes opportunities for fusion that would still be beneficial, particularly for AST optimising transformations.

We want to exploit fusion as much as possible, whilst still providing strong correctness guarantees. Therefore, in this chapter, we look at a different formalisation approach, that can handle the kind of tree transformations required in a compiler, and that takes a wider perspective on when fusion is successful.

5.1 What Do We Mean By Successful Fusion?

"The most important property of a program is whether it accomplishes the intention of its user." — C. A. R. Hoare [55]

Tree transformations, such as AST transforming compiler phases, are inevitably written to perform a given job. For instance, the transformations in Example 1 were intended to optimise away given syntactic patterns. It is therefore vital that, if separately run tree transformations accomplish their intended behaviours, then so does the result of fusing them.

To preserve the intended behaviour of tree transformations, under fusion, it is sufficient to preserve all observable behaviour. A common interpretation of successful fusion is that the observable behaviour of fused transformations should be identical to that of the same transformations running individually one after the other. However, Example 1, on page 12, demonstrates that there

are cases which this does not capture. We are perhaps being overly conservative in trying to preserve behaviour that was never necessarily wanted.

It is not always possible to accurately determine the intentions of a developer from the code that they produce. In Dotty miniphases [1] postconditions are implemented by compiler developers, to encode the intended behaviour of a tree transformation. Checks during testing then ensure that they are preserved by fusion. Analogously, we will think of *fusibility* in terms of fusion that preserves relevant postconditions, as defined informally below.

Definition 1. Let `f1` and `f2` be tree transformations, such that the trees they return always satisfy a given postcondition, `p1` and `p2` respectively. We say that `f1` and `f2` can be successfully fused, with respect to `p1` and `p2`, if `fused f1 f2` also ensures both `p1` and `p2`. We refer to this as *postcondition-preserving fusion.*

This definition parameterises the success of fusion over specific postconditions. Thus, we ensure that the required behaviour of our tree transformations will be preserved. There may be cases in which two postconditions clearly cannot both be satisfied, for example if they are mutually exclusive. However, as compiler phases are generally working together towards the same endpoint, in reality, cases such as mutual exclusivity are unlikely.

Moreover, the postconditions specify exactly what the required behaviour is. If a given postcondition poses an obstacle to fusing transformations, it could, in some cases, be possible for the compiler developer to refine their expectations and write a new, less ambitious, postcondition to reflect this. However, the process of weakening a postcondition in this way is difficult to formalise, as it depends heavily on the intent and intuition of the developer.

Here we will consider purely functional tree transformations, that is functions that take a tree and return another tree, without any side-effects. Hence, we need only look at the inputs and outputs of our transformations, when examining behaviours. This means that postconditions can be specified as predicates on trees.

5.2 Defining Transformations & Postconditions

Here we explain how we have formalised and verified our criteria for successful fusion, illustrating this with a running example. We have also mechanised work from the remainder of this chapter, using the Coq proof assistant.

5.2.1 Defining tree structures.

To begin with, we need to define the tree structure that will represent ASTs in this work. Our tree definition is parameterised over a data type `X` representing node labels. To highlight the directly inductive nature of the following definitions, we differentiate between a `Leaf` and an inner `Node`, although this is not strictly necessary as a `Leaf` is just a `Node` with an empty list of children.

Definition 2. We define a `Tree` as: a `Leaf` labelled from some set of labels, or a `Node` with a label and a child list.

```
Tree := Leaf X | Node X (List Tree)
```

for some label type `X`.

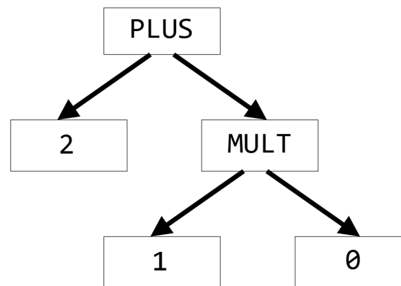

Figure 5.1: An AST example.

Example 2. We use the following language for our running example, to perform

program transformations on.

$$s ::= \texttt{if}\ (e)\ \texttt{then}\ \{s_1\}\ \texttt{else}\ \{s_2\}$$

$$e ::= n \mid b \mid e_1 + e_2 \mid e_1 * e_2 \mid \ \ldots$$

$$b ::= \texttt{true} \mid \texttt{false}$$

$$n ::= 0 \mid 1 \mid 2 \mid \ \ldots$$

Therefore, we can define a set of corresponding AST node labels that includes the following:

```
X := NAT Nat | BOOL Bool | IF | PLUS | MULT | ...
```

Note that labels `NAT` and `BOOL` are parameterised over sets of natural numbers and Boolean values respectively. For instance, `NAT 20` would represent the number 20.

Using the node labels in `X`, we can specify ASTs inductively, as in Definition 2. For example, the AST in Figure 5.1 can be specified as:

```
Node PLUS [Leaf (NAT 2), Node MULT [Leaf (NAT 1), Leaf (NAT 0)]].
```

5.2.2 Defining tree transformations.

Next, we formally define what we mean by a tree transformation. As in the miniphase framework, we separate the rewrite rules from the process of traversing and transforming the tree. This allows us to impose a standardised postorder traversal, and hence automatically fuse our transformations.

Definition 3. Let `f : Tree` $\rightarrow$ `Tree` be a function on trees. We define a function `transform` that takes `f` and applies it recursively to a given `Tree`.

```
transform f (Leaf x) := f (Leaf x)
transform f (Node x cs) := f (Node x (map (transform f) cs)).
```

Note that, as in many functional languages, `map` applies a given function to each element in a given list, and returns a corresponding list of the results.

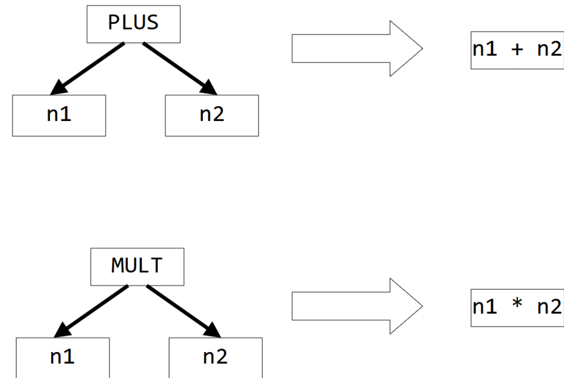

Figure 5.2: `arith_fold`: constant folding for given arithmetic expressions on ASTs.

Example 3. In Definition 3 we refer to `f : Tree` $\rightarrow$ `Tree`, which could be any function on trees. In the context of ASTs and compilation, this can be a rewriting of trees that translates syntactic elements or optimises the underlying program. Take, for example, `arith_fold`, as illustrated in Figure 5.2, which implements a constant folding optimisation for certain addition and multiplication. We can specifiy this function in a functional-style pseudocode.

```
arith_fold (Node PLUS [Leaf (NAT n1), Leaf (NAT n2)])
            := Leaf (NAT (n1 + n2))
arith_fold (Node MULT [Leaf (NAT n1), Leaf (NAT n2)])
            := Leaf (NAT (n1 * n2))
arith_fold t := t
```

We give the function in terms of pattern matching on the argument, as can be done in languages like Haskell. The function definition can be interpreted by comparing the given argument against each case in turn, until a match is found. For instance:

```
arith_fold (Node MULT [Leaf (NAT 2), Leaf (NAT 3)]) = Leaf (NAT 6)
```

and:

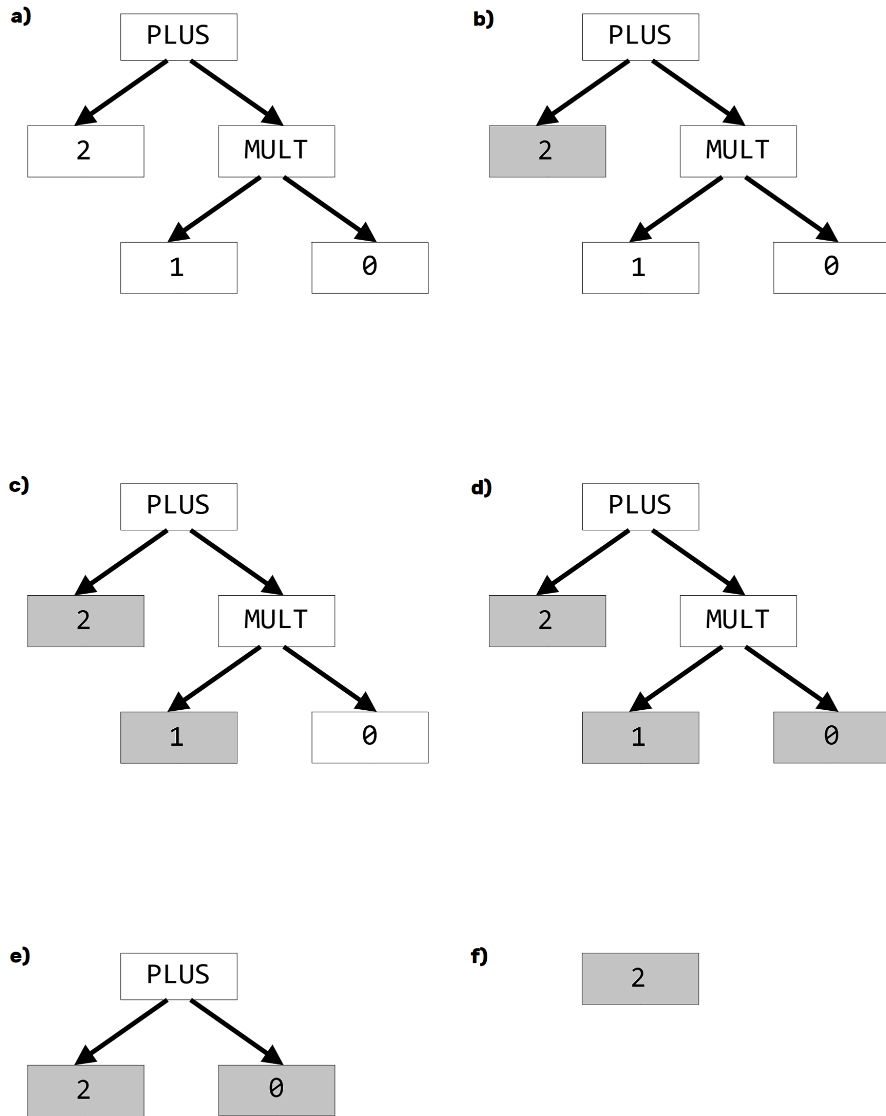

Figure 5.3: Transforming an AST with `arith_fold` on a postorder traversal.

```
arith_fold (Leaf (NAT 2)) = Leaf (NAT 2).
```

Consider the tree in Figure 5.1, namely:

```
Node PLUS [Leaf (NAT 2), Node MULT [Leaf (NAT 1), Leaf (NAT 0)]].
```

Applying `arith_fold` directly to this tree would have no effect, as the tree would only match with the final case of the `arith_fold` definition, and hence be returned unchanged. The solution is to apply `arith_fold` recursively instead, as in Figure 5.3, and this is where `transform` is used, to implement a postorder tree traversal.

5.2.3 Defining fusion of tree transformations.

We then define the process of fusion, taking two rewrite functions and applying both to each node visited, during a postorder `Tree` traversal.

Definition 4. For rewrite functions `f1, f2 : Tree` $\rightarrow$ `Tree` and `Tree t`, we define `fused` as:

```
fused f1 f2 t := transform (f2 ∘ f1) t
```

where $\circ$ is standard function composition.

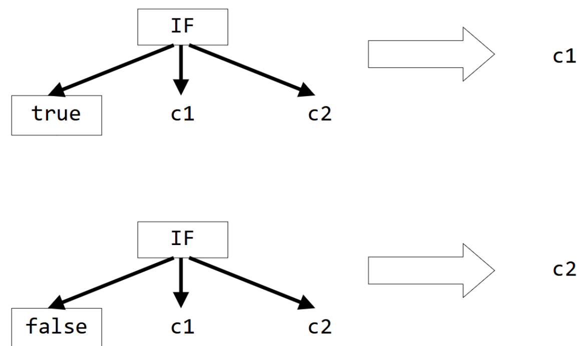

Figure 5.4: `if_fold`: constant folding for if expressions on ASTs.

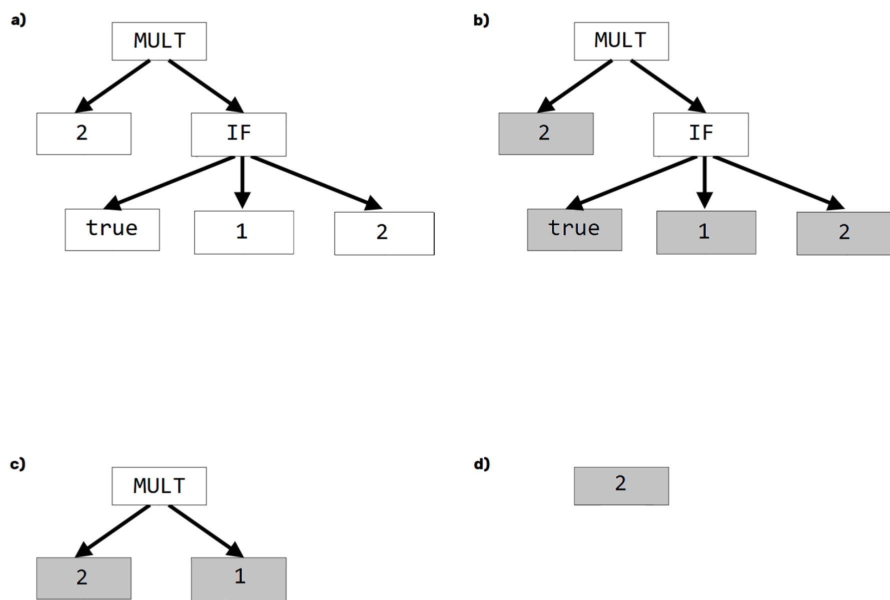

Figure 5.5: Transforming an AST with `fused arith_fold if_fold`.

Example 4. Consider the optimisation `if_fold`, illustrated in Figure 5.4, which performs constant folding on if expressions where the condition is a known Boolean value. This can be defined in pseudocode as:

```
if_fold (Node IF [Leaf (BOOL true), c1, c2])  := c1
if_fold (Node IF [Leaf (BOOL false), c1, c2]) := c2
if_fold t                                     := t
```

Suppose that we want to apply both optimisations, `arith_fold` (see Example 3) and `if_fold`, during our compilation process. Given that `arith_fold` already deals with folding addition and multiplication, we could have added some more cases to the definition that deal with if expressions. However, it then becomes more difficult to toggle off only the `if_fold` or to alter its implementation.

Instead, we can use miniphase style fusion to automatically create a singe tree transformation from these two separate definitions. From Definition 4, we get: `fused arith_fold if_fold = transform (if_fold` ∘ `arith_fold)`. Thus, by fusing `arith_fold` and `if_fold`, we get a tree transformation that performs a postorder tree traversal, applying `arith_fold` and then `if_fold` at each node visited, as in Figure 5.5.

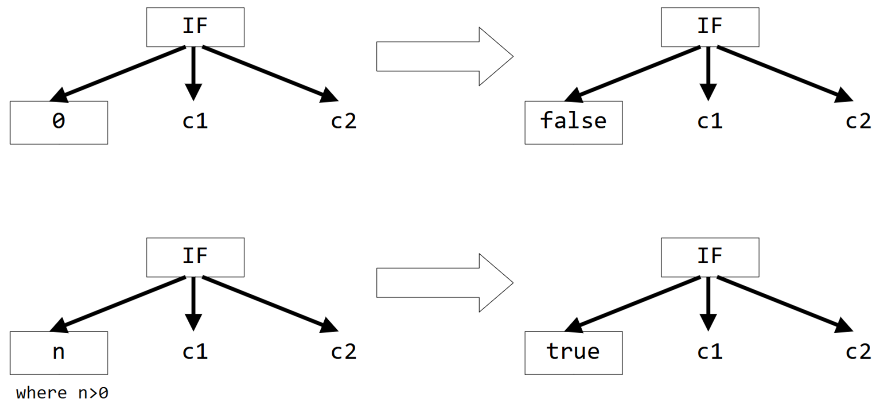

Figure 5.6: `int_to_bool`: A translation from integers to Booleans in ASTs.

5.2.4 Does the order of fusion matter?

Although we are fusing tree transformations, to apply in a single traversal, we still need to think about the order in which they are fused. The order of fusion dictates the order in which they are applied to each node visited. This means that fusing transformations in different orders can still produce different results.

Consider the following tree transformation, as illustrated in Figure 5.6. Suppose that the compiler's source language allows 0 to be evaluated as false and all other integers as true, but the target language does not. Then the compiler needs to translate any integers used as truth values into the respective Boolean, and enforce the fact that truth values must be Boolean. Hence, we can write a transformation to identify and rewrite such cases. We define a `Tree` rewrite function `int_to_bool` to implement this translation for if expressions.

```
int_to_bool (Node IF [Leaf (NAT 0), c1, c2])
              := Node IF [Leaf (BOOL false), c1, c2]
int_to_bool (Node IF [Leaf (NAT n), c1, c2])
              := Node IF [Leaf (BOOL true), c1, c2]
int_to_bool t := t
```

Suppose that we wish to fuse `int_to_bool` and `if_fold`, as defined in Example 4. Given that they both rewrite `IF` nodes, they will directly interact, and therefore the order of fusion is particularly significant. In Figure 5.7, transforming the example AST with `fused int_to_bool if_fold` will result in a partly optimised graph. However, if the order of fusion is reversed, then the `if_fold` transformation will not be able to optimise at any node.

5.2.5 Defining postcondition checks.

Now we can think about postconditions for our transformations. We define postconditions as predicates over `Trees`, allowing us to express some `Tree` property that must hold after a tree has been transformed. Particularly with AST transformations in compilers, we would commonly want such a property to hold for

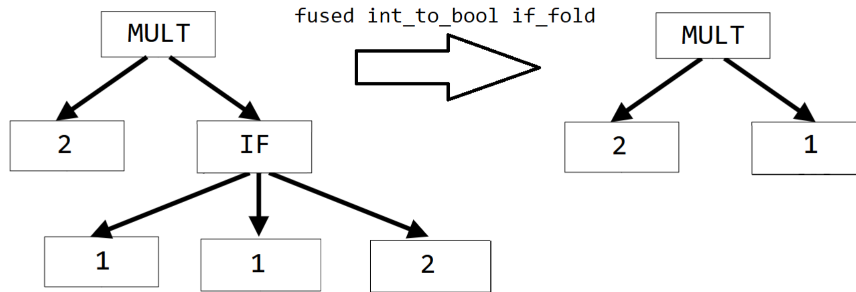

Figure 5.7: Transforming an AST with `fused int_to_bool if_fold`.

the entire `Tree`. Hence, we define a function to check recursively that some predicate holds for a node and all of its descendants.

Definition 5. Let `p : Tree` $\rightarrow$ {`True, False`} be a predicate. We define a function to check it recursively:

```
check p (Leaf x) := p (Leaf x)
check p (Node x cs) := p (Node x cs) ∧ ∀ c ∈ cs, check p c
```

Example 5. We can write postcondition predicates for the example optimisations that we have already defined. For instance, the purpose of `arith_fold` is to calculate addition and multiplication of constants at compile time. So, a potential postcondition is that there are no more such nodes that could be rewritten. This can be expressed as the following function:

```
p_arith_fold (Node PLUS [Leaf (NAT _), Leaf (NAT _)] = False
p_arith_fold (Node MULT [Leaf (NAT _), Leaf (NAT _)] = False
p_arith_fold _                                      = True
```

Predicate `p_arith_fold` corresponds to the idea that function `arith_fold` will make no further change. However, like `arith_fold`, `p_arith_fold` is dealing with a single node rather than the entire tree. Therefore, `check`, as specified in Definition 5, can be used to check `p_arith_fold` recursively across the tree, analogously to the way that `transform` applies a tree transformation.

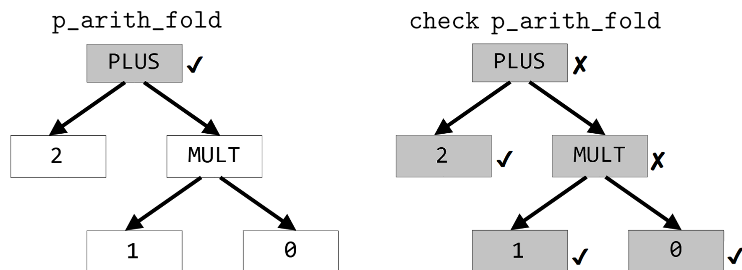

Figure 5.8: Evaluating a predicate on an AST, both directly and using `check` to check recursively.

Consider Figure 5.8, in which the predicate `p_arith_fold` is being evaluated on an AST, both directly and using `check` to check recursively. The nodes are shaded if they have been visited, and annotated with a check mark ✓ if `p_arith_fold` evaluates to `True` at that node, or a cross mark ✗ if it evaluates to `False`. The AST satisfies `p_arith_fold`, when evaluated directly. However when checked recursively, using `check` to perform a postorder traversal, the postcondition is not satisfied. This reflects the fact that `arith_fold` cannot make further changes, to the example AST, but `transform arith_fold` can.

5.2.6 Tree transformations and their postconditions.

We use postconditions to describe what their related transformations will achieve. Therefore, we need to assume that an individual tree transformation will ensure its related postcondition. If this is not the case before fusion, then its behaviour after fusion cannot reasonably be expected to be correct either. It is the compiler developer's job to confirm that the postcondition that they provide is appropriate for the corresponding transformation.

Given that all of our rewrite rules are applied recursively, we can assume that all descendants of a node being rewritten have already been rewritten and hence already satisfy the postcondition. Therefore, we say that a rewrite function `f` `ensures` a postcondition `p` if, for any tree `t` whose descendants satisfy `p`, the

result of rewriting `t` will also satisfy `p`.

Definition 6. For some function `f : Tree` $\rightarrow$ `Tree` and postcondition `p : Tree` $\rightarrow$ {`True, False`}, we define `ensures` as: `ensures f p :=`

$$\forall\ \texttt{t : Tree, (}\forall\ \texttt{c} \in \texttt{children t, check p c)} \Rightarrow \texttt{check p (f t)}$$

Example 6. Consider the example tree transformation `arith_fold`, as defined in Example 3, and corresponding postcondition `p_arith_fold`, as defined in Example 5. We know informally, as the compiler developer, that `p_arith_fold` is an appropriate postcondition to express the desired outcome of `arith_fold`. Using `ensures`, we can state this knowledge formally, in a way that can be verified: `ensures arith_fold p_arith_fold`.

5.3 Fusing Pairs of Transformations

Having established that postconditions are appropriate for their respective tree transformations, we define the necessary relationships between a transformation and the postcondition of the other transformation that it is being fused with. This relationship is not symmetric, differing depending on whether the transformation is the first or second of the fused pair.

Consider arbitrary tree transformations `f1` and `f2`, each of which ensures a postcondition, `p1` and `p2` respectively. Suppose that we want to fuse these tree transformations, but also to make sure that both `p1` and `p2` will hold at the end of running the fused transformation. It is not practical to expect that the two transformations will be entirely independent. Therefore we need to guarantee that the interaction between fused transformations will be safe with respect to preserving postconditions. We can motivate two criteria which will be sufficient to ensure postcondition-preserving fusion, as follows.

Fusion Criterion 1. As we visit a node, on our postorder traversal, we first rewrite it with `f1`. If we have defined our transformation and postcondition correctly, then we know that `p1` should hold after this rewrite. Next we have to

rewrite the node with `f2`. To be sure that `p1` will still hold after this rewrite, we need to know that `f2` preserves it, as in Figure 5.9.

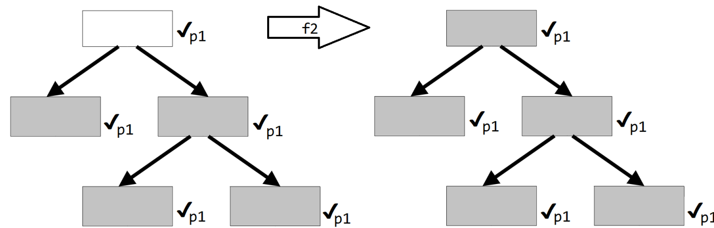

Figure 5.9: Illustrating Fusion Criterion 1 (`FC1`)

Fusion Criterion 2. When applying `f2` individually, any children of a node that it rewrites have just been rewritten by `f2` themselves, so we can assume that they already satisfy `p2`. If we are applying `fused f1 f2` instead, the node will first be rewritten by `f1`. So, we need to know that `f1` preserves `p2` for all children of the nodes that it rewrites, as in Figure 5.10.

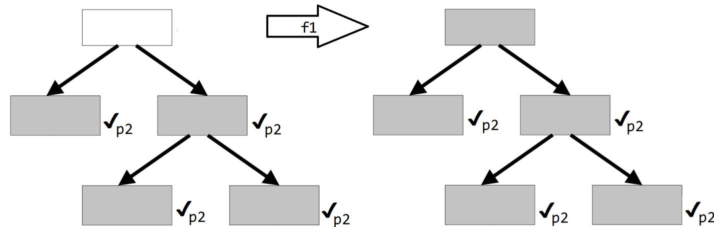

Figure 5.10: Illustrating Fusion Criterion 2 (`FC2`)

Hence, formally, our fusion criteria for postcondition-preservation are defined as follows.

Definition 7. For some rewrite function `f` and postcondition `p`, we have two

criteria:

```
FC1 f p := ∀ t, check p t ⇒ check p (f t)
FC2 f p := ∀ t, (∀ c ∈ children t, check p c)
                         ⇒ ∀ c′ ∈ children (f t), check p c′
```

These definitions can then be used to express criteria that ensure that fusing two tree transformations will preserve given corresponding postconditions. Suppose we are fusing `f1` and `f2`, in that order, with postcondition `p1` and `p2` respectively. We require `FC1 f2 p1` so that we know `f2` preserves `p1`. We also require `FC2 f1 p2` so that we know that `f1` preserves `p2` in any children of the node that it is rewriting.

Therefore, with our criteria, we can assemble our theorem for postcondition-preserving fusion of tree transformations.

Theorem 1. Let `f1` and `f2` be tree rewrite functions, and `p1` and `p2` be postcondition predicates. Then:

```
ensures f1 p1 ∧ ensures f2 p2 ∧ FC1 f2 p1 ∧ FC2 f1 p2

⇒ ∀ t, check p1 (fused f1 f2 t) ∧ check p2 (fused f1 f2 t)
```

In proving this theorem, and hence verifying our fusion criteria, we can split it into two lemmas, each considering a different postcondition.

Lemma 1. Let `f1` and `f2` be rewrite functions and `p1` a postcondition. Then:

```
ensures f1 p1 ∧ FC1 f2 p1 ⇒ ∀ t, check p1 (fused f1 f2 t)
```

Proof. We prove this lemma by induction, on the structure of the `Tree` data type.

Let `t = Leaf x`. Then `fused f1 f2 t` becomes `f2 (f1 t)`. We know that `f1 t` ensures `check p1`, due to `ensures f1 p1`. Hence, `f2 (f1 t)` also satisfies `check p1`, as `FC1` requires that `f2` preserves `p1`.

Now, let `t = Node x cs`. Then `fused f1 f2 t` becomes `f2 (f1 (Node x (map (fused f1 f2) cs)))`. Our induction hypothesis is that `check p1`

(fused f1 f2 c), for all c in cs. Therefore, f1 (Node x (map (fused f1 f2) cs)) satisfies check p1, due to ensures f1 p1. And so, as before, FC1 ensures that f2 (f1 (Node x (map (fused f1 f2) cs))) also satisfies check p1. □

Lemma 2. Let f1 and f2 be rewrite functions and p2 a postcondition. Then:

$$\texttt{satisfies f2 p2} \land \texttt{FC2 f1 p2} \Rightarrow \forall\ \texttt{t, check p2 (fused f1 f2 t)}$$

Proof. As with the previous lemma, we proceed by induction.

Let t = Leaf x. Then fused f1 f2 t becomes f2 (f1 t). Since t has no children, FC2 dictates that any children of f1 t must satisfy check p2. Therefore, f2 (f1 t) satisfies check p2, due to ensures f2 p2.

Now, let t = Node x cs. Then fused f1 f2 t becomes f2 (f1 (Node x (map (fused f1 f2) cs))). Our induction hypothesis is that check p2 (fused f1 f2 c), for all c in cs. So, any children of f1 (Node x (map (fused f1 f2) cs)) must satisfy check p2, due to FC2. Therefore, we know that f2 (f1 (Node x (map (fused f1 f2) cs))) satisfies check p2, since we know ensures f2 p2. □

Unit laws and identities.

In this subsection, we detail some unit laws that relate to specialised cases of tree rewrite functions and postconditions. In particular, we look at cases in which the rewrite function is the identity function, and also cases in which the relevant postcondition is always true.

Definition 8. Let id : Tree → Tree be the identity tree rewrite function. That is, id t := t, for any Tree t.

Lemma 3. Let f be a tree rewrite function and p a postcondition. Then:

$$\texttt{ensures f p} \Rightarrow \forall\ \texttt{t, check p (fused id f t)}$$

Lemma 4. Let f be a tree rewrite function and p a postcondition. Then:

$$\texttt{ensures f p} \Rightarrow \forall\ \texttt{t}, \texttt{check p (fused f id t)}$$

Definition 9. We define `alwaysTrue : (Tree → Prop) → Prop` to be a predicate denoting that a postcondition evaluates to true for all `t : Tree`. That is, for `p : Tree → Prop`:

$$\texttt{alwaysTrue p := } \forall\ \texttt{t, p t = True}.$$

Lemma 5. Let `f1` and `f2` be tree rewrite functions and `p` a postcondition. Then:

$$\texttt{alwaysTrue p} \Rightarrow \forall\ \texttt{t}, \texttt{check p (fused f1 f2 t)}$$

The unit laws in this subsection are fairly trivial. However, they are included for the sake of completeness, and to illustrate some cases in which the previously defined criteria `FC1` and `FC2` are obsolete in proving that fusion will be successful.

Generalising proofs for similar examples.

It is possible to prove that rewrite rules will always satisfy the criteria, if they have some other property which is even easier to check. Making generalisations like this can further reduce the effort required in verification or testing. For example, in this section we prove that a rewrite function which just acts to remove a section of the tree will satisfy our fusion criteria with respect to any postcondition.

Take, for instance, the `if_fold` transformation, which only acts to remove a part of the tree. This is essentially performing the same operation as a multitude of other optimisations that are eliminating dead or unnecessary subgraphs. Therefore, the shape of the functions and the shape of the proofs of fusibility criteria will ineveitably be similar. It is beneficial to be able to generalise and reuse such proofs, rather than entirely repeating them.

We will call a rewrite function a removal if rewriting a tree always returns either the tree itself or one of its children. Hence, the transformation is simply removing part of the tree, if it changes the tree at all. This is the case for

the example transformation `if_fold` and for other AST optimisations such as folding Boolean expressions.

Definition 10. A rewrite function `f` is a *removal* if, for every `Tree t`, either `f t = t` or `f t` $\in$ `children t`. We will denote this property as: `removal f`.

We can prove that all removal transformations will satisfy both pairwise fusion criteria `FC1` and `FC2`, for all possible postconditions. This mostly just falls out of the definitions, since everything is happening recursively.

Lemma 6. Let `f` be a rewrite function and `p` a postcondition. Then: `removal f` $\Rightarrow$ `FC1 f p`.

Proof. Let `t` be a `Tree` and suppose that `check p t` holds. We need to show that `check p (f t)` holds. We proceed by case analysis on `t`.

Let `t = Leaf x`. Due to `removal f`, we know that `f t` is either just `t` or a child of `t`. Since `t` is a `Leaf` and thus has no children, it must be the case that `f t = t`. And we already know that `check p t` holds.

Now, let `t = Node x cs`. If `f t = t`, then as before, we are done. Otherwise, `f t` $\in$ `children t`. Due to the recursive nature of `check`, we already know that `check p c` holds for any `c` $\in$ `children t`. $\square$

Lemma 7. Let `f` be a rewrite function and `p` a postcondition. Then: `removal f` $\Rightarrow$ `FC2 f p`.

Proof. Let `t` be a `Tree` and suppose that $\forall$ `c` $\in$ `children t, check p c`. We show $\forall$ `c'` $\in$ `children (f t), check p c'`, proceeding by case analysis on `t`.

Let `t = Leaf x`. As in the previous lemma, since `t` has no children, it must be the case that `f t = t`. And we have already assumed that $\forall$ `c` $\in$ `children t, check p c`.

Now, let `t = Node x cs`. Again, if `f t = t`, we are done. Otherwise, `f t` $\in$ `children t` and the recursive definition of `check` grants us what we seek. $\square$

Checking that `removal f` holds should be straightforward and directly relates to the implementation of any given rewrite function. By generalising our

proofs, in this way, we can yield strong guarantees without having to write repetitive and similar proofs for similar tree transformations.

5.4 Fusing Three or More Transformations

In order to fully reap the benefits of fusion, we want to be able to fuse a list of multiple transformations. Hence, in this section we extend our pairwise results to consider a list of arbitrary-many tree transformations to fuse.

We define `compose_list`, which recursively composes a list of tree rewrite functions, and which we can use in place of `compose` in our definition `fused_list`.

Definition 11. For a list of rewrite functions and a `Tree t`, we define:

```
compose_list [] t := t
compose_list (f :: fs) t := compose_list fs (f t)
```

Definition 12. For some list of rewrite functions `fs` and `Tree t`, we define:

```
fused_list fs t := transform (compose_list fs) t
```

Now that we can fuse multiple transformations together, we need to consider how our fusion criteria will scale. We choose an arbitrary rewrite function `f`, from our list, with its corresponding postcondition `p`. Using `FC1` and `FC2`, we can split our list into three parts `before ++ [f] ++ after` and derive separate criteria for the transformations before and after `f` in the fusion order.

A transformation after `f` must preserve `p` so as not to undo the work that `f` has done. So all transformations after `f` must satisfy `FC1` with respect to `p`. A transformation before `f` must not introduce children that violate `p` to ensure that `f` can satisfy `p`. So all transformations before `f` must satisfy `FC2` with respect to `p`. Hence we define below our extended criteria.

Definition 13. For some list of rewrite functions `fs` and postcondition `p`, we have two extended fusion criteria:

```
after_FC1 p fs := ∀ f ∈ fs, FC1 f p
before_FC2 p fs := ∀ f ∈ fs, FC2 f p
```

The following theorem states that, for any given transformation in our list, if it ensures its corresponding postcondition, and both extended fusion criteria are satisfied, then the result of applying our fused list will satisfy that postcondition.

Theorem 2. Let `fs ++ [f] ++ fs′` be a list of rewrite functions and `p` a postcondition. Then:

$$\texttt{ensures f p} \wedge \texttt{before_FC2 p fs} \wedge \texttt{after_FC1 p fs}'$$
$$\Rightarrow \forall\ \texttt{t, check p (fused_list (fs ++ [f] ++ fs}'\texttt{) t)}$$

The proof of this theorem proceeds by nested induction on various parts of the list, and on the `Tree` data type structure. We split this into a number of lemmas that we will prove before returning to the proof of Theorem 2.

To start with, we prove that a composed list of tree rewrite functions, used to rewrite a tree node, will preserve a postcondition `p` on the tree, as long as the composed functions satisfy criterion `after_FC1` with respect to `p`. The criterion `after_FC1` states that each of the composed functions will individually preserve `p`. Hence, the proof follows naturally from the definition of `after_FC1`.

Lemma 8. Let `t` be a `Tree`, `p` a postcondition, and `fs` a list of rewrite rules. Then:

$$\texttt{check p t} \wedge \texttt{after_FC1 p fs} \Rightarrow \texttt{check p (compose_list fs t)}$$

Proof. We prove this lemma by induction on `fs`. If `fs` is an empty list then `compose_list fs t` is just `t`, and we are done.

Let `fs = f :: fs′`. Then `compose_list fs t` becomes `compose_list fs′ (f t)`. Our induction hypothesis is, for any `Tree t′`, `check p t′` $\Rightarrow$ `check p (compose_list fs′ t′)`. Hence, we just need to prove `check p (f t)`, which directly follows from `after_FC1 p (f :: fs′)`. □

In the next lemma we prove that, if the first tree rewrite function in a fused list ensures a given postcondition `p`, and all subsequent functions in the list preserve `p`, then the result of applying the fused list to any `Tree` will satisfy `p`.

Lemma 9. Let `[f] ++ fs` be a list of rewrite functions and `p` a postcondition. Then:

$$\texttt{ensures f p} \land \texttt{after_FC1 p fs}$$

$$\Rightarrow \forall\ \texttt{t, check p (fused_list ([f] ++ fs) t)}$$

Proof. We begin by induction on the list `fs`.

Let `fs` be an empty list. Then `fused_list ([f] ++ fs) t` becomes `transform f t`. We proceed by induction on `t`. If `t = Leaf x`, then `transform f t` is just `f t`, and `check p (f t)` follows from `ensures f p`, as `t` has no children.

Otherwise, we have `t = Node x cs`, and so `transform f t` becomes `f (Node x (map (transform f) cs))`. Our induction hypothesis is that, for all `c` in `cs`, `check p (transform f c)`. So, we have `check p (f (Node x (map (transform f) cs)))`, again from `ensures f p`.

Now, let `fs = fs′ ++ [f′]`. Our induction hypothesis is, for any `Tree t′`, `check p (fused_list ([f] ++ fs′) t′)`. We proceed by induction on `t`.

Let `t = Leaf x`. Then `fused_list ([f] ++ fs) t` becomes `f′ (compose_list fs′ (f t))`. From the `fs` induction hypothesis, we know `check p (compose_list fs′ (f t))` holds. From `after_FC1 p fs` we have `FC1 f′ p`. Therefore, we have `check p (f′ (compose_list fs′ (f t)))`.

Now, let `t = Node x cs`. Then `fused_list ([f] ++ fs) t` becomes `f′ (compose_list fs′ (f (Node x (map (fused_list ([f] ++ fs)) cs))))`. Our induction hypothesis for `t` is that `check p (fused_list ([f] ++ fs) c)` holds, for all `c` in `cs`. So, `check p (f (Node x (map (fused_list ([f] ++ fs)) cs)))` follows from `ensures f p`. Thus, from Lemma 8, we have `check p (compose_list fs′ (f (Node x (map (fused_list ([f] ++ fs)) cs))))`. Finally, from `after_FC1 p fs` we have `FC1 f′ p`, and so `f′` preserves `check p` and we are done. □

Our final lemma, before constructing the proof of Theorem 2, considers the preservation of a postcondition in the descendants of a given node. Namely, we prove that, if the tree rewrite functions in a composed list would individually

preserve postcondition `p` in a node's descendants, and the descendants of a given node satisfy `p`, then rewriting the given node with the composed function list will result in a node with descendants that still satisfy `p`.

Lemma 10. Let `t` be a `Tree`, `p` a postcondition, and `fs` a list of rewrite rules. Then:

$$(\forall\ \texttt{c} \in \texttt{children t, check p c}) \wedge \texttt{before_FC2 p fs}$$
$$\Rightarrow \forall\ \texttt{c}' \in \texttt{children (compose_list fs t), check p c}'$$

Proof. We prove this lemma by induction on `fs`. If `fs` is an empty list then `compose_list fs t` is just `t`, and we are done.

Let `fs = f :: fs`′. Then `compose_list fs t` becomes `compose_list fs`′ `(f t)`. Our induction hypothesis is, for any `Tree t`′, we have $(\forall$ `c` $\in$ `children t`′, `check p c`) $\Rightarrow \forall$ `c`′ $\in$ `children (compose_list fs`′ `t`′`), check p c`′. Hence, we just need to show $\forall$ `c` $\in$ `children (f t), check p c`, which follows directly from `before_FC2 p (f :: fs`′`)`. □

Now we can use Lemmas 8, 9 and 10, to prove Theorem 2, which states that extended fusion criteria `after_FC1` and `before_FC2` are sufficient to ensure that a given postcondition is preserved by an arbitrary length list of fused tree transformations.

Proof of Theorem 2. Let `fs ++ [f] ++ fs`′ be a list of rewrite functions and `p` a postcondition. Suppose: `ensures f p` and `before_FC2 p fs` and `after_FC1 p fs`′. We want to show that:

$$\forall\ \texttt{t, check p (fused_list (fs ++ [f] ++ fs}'\texttt{) t)}.$$

We begin by case analysis on the list `fs`. If `fs` is the empty list, then we are just considering functions `[f] ++ fs`′, which is exactly the case covered by Lemma 9.

Instead, let `fs = g :: gs`. We proceed by induction on `t`.

Let `t = Leaf x`. Then we know that `fused_list (fs ++ [f] ++ fs`′`) t` is `compose_list hs (f (compose_list gs (g t)))`. Since `t` is a `Leaf`, and

consequently has no children, we can say that ∀ `c` ∈ `children t, check p t`. Therefore, we have ∀ `c` ∈ `children (compose_list gs (g t)), check p t`, due to Lemma 10 and `before_FC2 p fs`. From `ensures f p`, we then have `check p (f (compose_list gs (g t)))`. And, `check p (compose_list fs′ (f (compose_list gs (g t))))` follows from Lemma 8.

Let `t = Node x cs`. Then `fused_list (fs ++ [f] ++ fs′) t` becomes `compose_list fs′ (f (compose_list gs (g t′)))`, where `t′ = Node x (map (fused_list (fs ++ [f] ++ fs′)) cs)`. Our induction hypothesis is that, for all `c` in `cs`, we know `check p (fused_list (fs ++ [f] ++ fs′) c)` holds. Thus, we can follow the same reasoning as the `Leaf` case to complete the proof. □

5.5 Fusing Blocks of Fused Transformations

In the previous section, we extended the results for fusing two transformations, to look at fusing a list of arbitrary many tree transformations. That approach considered each transformation on its own, when defining postconditions and criteria for successful fusion. However, it may also be useful to examine fusion at the level of blocks, consisting of transformations that have already been fused. In this section, we analyse how our previous results apply in such a case.

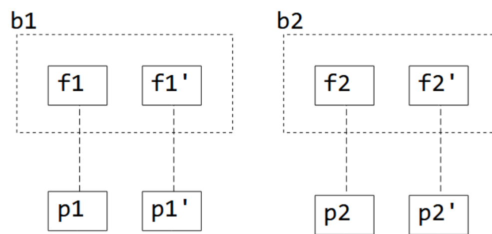

Figure 5.11: Blocks of fused transformations.

5.5.1 Considering blocks as tree transformations.

The first thing to note is that we can view these blocks in terms of the tree transformations themselves. Suppose that `b1` and `b2` are blocks of fused tree transformations, defined as follows.

```
b1 := fused f1 f1′ = transform (f1′ ∘ f1)
b2 := fused f2 f2′ = transform (f2′ ∘ f2)
```

Since $\circ$ is standard function composition, we know that `f1′ ∘ f1` and `f2′ ∘ f2` both have type `Tree → Tree`. Hence, we can use them as tree rewrite functions to plug into our previous results.

Similarly, suppose postconditions `p1`, `p1′`, `p2` and `p2′` have been defined for rewrite functions `f1`, `f1′`, `f2` and `f2′` respectively. Then we can obtain a single postcondition for each block. For instance, `fun t => p1 t ∧ p1′ t` would then be a suitable postcondition for `f1′ ∘ f1` in block `b1`.

From these observations, we can plug the components of blocks `b1` and `b2` into Theorem 1 as follows.

```
  ensures (f1′ ∘ f1) (fun t => p1 t ∧ p1′ t)
∧ ensures (f2′ ∘ f2) (fun t => p2 t ∧ p2′ t)
  ∧ FC1 (f2′ ∘ f2) (fun t => p1 t ∧ p1′ t)
  ∧ FC2 (f1′ ∘ f1) (fun t => p2 t ∧ p2′ t)

⇒ ∀ t, check (fun t => p1 t ∧ p1′ t) (fused (f1′ ∘ f1) (f2′ ∘ f2) t)
    ∧ check (fun t => p2 t ∧ p2′ t) (fused (f1′ ∘ f1) (f2′ ∘ f2) t)
```

Hence, in the resulting formalisation, the fusion of `b1` and `b2` boils down to:

```
fused (f1′ ∘ f1) (f2′ ∘ f2)
```

or equivalently:

```
transform (f2′ ∘ f2 ∘ f1′ ∘ f1).
```

5.5.2 Reusing established properties.

One of the key benefits of a modular approach is the ability to reuse components. In this block-based situation, if we have already reasoned about fusion *within* the blocks, then we have established some useful relationships. These can be reused as stepping stones when proving the required hypotheses for successful fusion *between* the blocks.

For instance, suppose that in block `b1` we have proved `f1` and $\texttt{f1}'$ can be successfully fused with respect to corresponding postconditions `p1` and $\texttt{p1}'$. If this has been done using the criteria in Theorem 1, then we have previously established both $\texttt{FC1 f1}'\ \texttt{p1}$ and $\texttt{FC2 f1 p1}'$. Using this, we already know that $\texttt{f1}' \circ \texttt{f1}$ ensures both `p1` and $\texttt{p1}'$, as is detailed in the following two lemmas.

Lemma 11. Let `f1` and $\texttt{f1}'$ be tree rewrite functions and `p1` a postcondition. Then:

$$\texttt{ensures f1 p1} \wedge \texttt{FC1 f1}'\ \texttt{p1} \Rightarrow \texttt{ensures (f1}' \circ \texttt{f1) p1}$$

Lemma 12. Let `f1` and $\texttt{f1}'$ be tree rewrite functions and $\texttt{p1}'$ a postcondition. Then:

$$\texttt{ensures f1}'\ \texttt{p1}' \wedge \texttt{FC2 f1 p1}' \Rightarrow \texttt{ensures (f1}' \circ \texttt{f1) p1}'$$

Having the knowledge, then, that $\texttt{f1}' \circ \texttt{f1}$ ensures both `p1` and $\texttt{p1}'$, makes it trivial to establish: $\texttt{ensures (f1}' \circ \texttt{f1) (fun t => p1 t} \wedge \texttt{p1}'\ \texttt{t)}$, and an analogous approach can be taken for the components of `b2`. Thus it would only remain to check that: $\texttt{FC1 (f2}' \circ \texttt{f2) (fun t => p1 t} \wedge \texttt{p1}'\ \texttt{t)}$ and $\texttt{FC2 (f1}' \circ \texttt{f1) (fun t => p2 t} \wedge \texttt{p2}'\ \texttt{t)}$ hold.

5.5.3 How useful is this block-based approach?

One problem with viewing fused transformations as blocks in this way is that the resulting transformations and their postconditions grow increasingly complicated. Thus it can be increasingly difficult to reason about these components. It becomes necessary to consider $\texttt{f1}' \circ \texttt{f1}$ and $\texttt{fun t => p1 t} \wedge \texttt{p1}'\ \texttt{t}$ rather

than `f1`, `f1`$'$, `p1`, and `p1`$'$ separately. In this sense, some of the benefits of modularity are lost. Instead it may be more useful to take the view outlined in Section 5.4 and continue to consider the originally defined components individually.

In the end, one benefit of this framework is that the user could take either of these approaches: either cumulatively building components into more complex components, or continuing to view everything as collections of very simple components. Ultimately it depends on the user and use-case in question.

5.6 Preserving Correctness

Thus far we have considered tree transformation in terms of satisfying a developer prescribed postcondition. However, the usual metric for determining the correctness of compiler phases is to look at the preservation of program semantics. This is a more fundamental and nonnegotiable requirement for program transformations within the compiler. The postconditions that we have been focusing on are more a measure of effectiveness than correctness. For example, a postcondition that states that there should be no remaining instances of adding 0, says nothing about the correctness of compilation: the resulting AST could evaluate to something entirely different than the original AST.

Therefore, it is important to examine how the tree transformations and fusion framework that we have developed interact with compiler correctness. An essential stipulation for fusion of any compiler phases has to be that individually correct phases that are fused must remain correct. That is, as well as postcondition-preserving fusion, we need to have correctness-preserving fusion.

The approach that we take here is simplistic, and in line with the simple examples that we will consider in the next section. We look at an evaluation strategy for the program represented by an AST, whereby the value of each node depends only on the node label and the value of its children. There is certainly more sophisticated future work to be done, looking at preserving a more realistic operational semantics.

Where `X` is our arbitrary set of node labels, we assume that `leafVal` and

`nodeVal` are evaluation functions, mapping leaves and inner nodes, respectively, to an arbitrary set of possible values. We can then define a function `eval` that applies these mappings recursively to evaluate the AST.

Definition 14. Let `Value` be a set of possible values. And let `leafVal : X` $\rightarrow$ `Value` and `nodeVal : X` $\rightarrow$ `List Value` $\rightarrow$ `Value` be mappings from nodes to values. Then we can evaluate a `Tree` recursively as follows.

```
eval (Leaf y) := leafVal y
eval (Node x cs) := nodeVal x (map eval cs)
```

Having defined the evaluation mechanism, we define our notion of correctness as `preservesEval`. Essentially, the result of a correct program transformation must evaluate to the same value as the original program.

Definition 15. Let `f` be a `Tree` rewrite function. Then we define the predicate `preservesEval` as follows.

```
preservesEval f := ∀ t, eval (f t) = eval t
```

Finally we have all of the components to formulate a theorem stating that the result of fusing two evaluation-preserving tree transformations is also evaluation-preserving. In other words, fusion preserves `preservesEval`. The proof of this theorem is then fairly straightforward, via induction on tree structure.

Theorem 3. Let `f1` and `f2` be `Tree` rewrite functions, and `t` a `Tree`. Then:

```
preservesEval f1 ∧ preservesEval f2
  ⇒ eval (fused f1 f2 t) = eval t
```

Proof. This proof proceeds by induction over the structure of `Tree`.

Let `t = Leaf x`. Then `fused f1 f2 t` becomes `f2 (f1 t)`. We know that `eval (f1 t) = eval t`, due to `preservesEval f1`. Hence, `eval (f2 (f1 t)) = eval (f1 t) = eval t`, due to `preservesEval f2`.

Now, let `t = Node x cs`. Then `fused f1 f2 t` becomes `f2 (f1 (Node x (map (fused f1 f2) cs)))`. The induction hypothesis is that `eval (fused`

`f1 f2 c) = eval c`, for all `c` in `cs`. So, `map eval (map (fused f1 f2) cs) = map eval cs`.

Thus, `eval (Node x (map (fused f1 f2) cs) = nodeVal x (map eval (map (fused f1 f2 cs))) = nodeVal x (map eval cs) = eval (Node x cs)`. So, due to `preservesEval f1` and `preservesEval f2`, we have `eval (fused f1 f2 t) = eval (f2 (f1 (Node x (map (fused f1 f2) cs)))) = eval (Node x (map (fused f1 f2) cs)) = eval t`. □

Note that, it similarly follows that: `(fused f2 f1 t) = eval t`. This is because the premises simply state that both `f1` and `f2` preserve the value of the `Tree` they are transforming. Thus, the order of transformations, whilst possibly having an effect on the resulting `Tree`, will not have an effect on its value.

There are, of course, many nuances that this definition of correctness fails to account for. For example, where a programming language involves side effects, it is insufficient to look solely at the result of some such evaluation function. However, for the minimal, referentially transparent programs that we have been considering, this basic approach is a start.

5.7 Discussion

This chapter has detailed a method for modular reasoning about fusing modular tree transformations, such as those found in compiler phases. By imposing a uniform traversal order on these transformations, in this case postorder, it is possible to fuse them by interleaving, and thus reduce the number of tree traversals required. We specify that this fusion can only take place if it preserves the postconditions defined for each transformation by the compiler developer. This allows us to ensure that they still behave as required.

To decide whether a series of tree transformations is fusible, without fusing them, we have derived and verified a set of fusion criteria relating pairs of individual transformations and postconditions. These criteria can be checked prior to fusion, thus providing a modular approach. The benefits are that each

check or proof is simpler, and that the postconditions form an interface between transformations which allows for a degree of refactoring.

We have also used simple examples of program transformations, which translate given features of perform optimisations, to assess how applicable the work in this chapter is. These examples demonstrate that it is straightforward to implement realistically useful transformations within this framework. However, these transformations are simplistic in the scheme of real-world compiler phases, and as such it would be beneficial to consider some more complex examples. Moreover, the postconditions that we have implemented for optimisation tend to specify that no more rewriting could be done by that particular transformation. This does not capture ideas such as correctness and improvement, which would be interesting to also consider.

Chapter 6

Fusing Graph Transformations

In the previous chapter, we examined postcondition-preserving and correctness-preserving fusion of AST transformations. Now we look to extend these ideas, to apply to a wider range of typical compiler phases. As the Background chapter indicated, many types of graphs have been used for implementing compilers, with the AST and the CFG perhaps the most common. Therefore, it would be useful to broaden the criteria that we have developed accordingly, beyond solely considering trees.

Many compilers use more general graph structures for program representation, in addition to or instead of tree representations. This is because graphs with shared nodes and cycles can allow for better encoding of useful program information, such as explicit control flow. In turn, this extra information can facilitate more sophisticated optimisations. Moreover, avoiding the duplication of shared nodes leads to a more compact representation which can reduce storage requirements.

This chapter explains how the work in the previous chapter can be generalised to a less restricted graph structure. The first two sections outline the

possible limitations of AST transformations in a compiler, and the performance gains to be made in identifying and optimising loops. The remaining sections take the formalisation from the previous chapter and extend the specification to instead consider graphs in general, before evaluating whether the definitions remain applicable on various examples.

To achieve this, we start by exploring an inductive definition of graphs. Conceptually, it seems that it should be easy to transition from working on trees to working on other graphs, since they are apparently similar structures. However, the recursive definition of trees that underpins our work so far is not so obviously applicable when graphs contain shared nodes. Hence, we look to the inductive graph definition by Erwig [3], in order to make the adjustments from trees to graphs as straightforward as possible. This approach leads to modular criteria for reasoning about combining modular graph transformations. The recursive nature of the graph definition mirrors the functional style of the formalisation, in a way that is reasonably clear and elegant.

6.1 Limitations of the AST Representation

The AST is ubiquitous in compiler design, generally seen as the most appropriate representation for the frontend phases. However, very few compilers find it practical to use an AST all the way down. Indeed, whilst the Dotty miniphase compiler has a great number of AST transformations, it is only compiling as far as Java bytecode, which will usually be processed at a lower level by the JVM. There are several properties of the AST that make it less suited to some of the compilation process.

The information presented by the AST is strongly linked to program syntax, which can sometimes obscure the program semantics. Compared to the initial parse tree, the AST does contain a more focused selection of information, resolving some structural ambiguity and abstracting away syntactic elements that are only specific to the source language. However, for representing complicated expressions, the AST can still require a large amount of nodes. Moreover, there

are no shared nodes, leading to repeated subtrees and masking some useful optimisation targets.

In addition to this, the greater number of nodes results in greater memory requirements. The AST can contain information that stops being useful after the frontend of the compiler pipeline. There are more compact graph structures for directly representing the control flow and dependence information that the later phases require. Therefore, after the AST has served its primary purpose there may be other graphical intermediate representations that are better suited for the rest of compilation.

6.2 Optimising Loops

Loops can provide conspicuous and rewarding targets for compiler optimisations. There is potential for significant overhead performance costs for using loops, even when the implementation is relatively efficient. In addition to this, any inefficiencies inside the loop are multiplied by the number of times the loop is executed, which could be many. So, where the compiler can make relevant improvements there are major performance gains to be made.

By using a suitable graph representation, it is straightforward to identify the loops that are prime for optimisation, as they are encoded explicitly. In an AST, the only way to locate a loop, for example, is to search for an appropriately labelled node. There is no structural representation, in terms of edges, to indicate the presence or semantics of a loop. In a CFG, on the other hand, program loops are generally represented by loops in the graph. This makes them more apparent and exposes how they behave.

Another graphical intermediate representation that overtly pays attention to loops is the program expression graph (PEG) [23], as discussed in Section 2.2.3. Similarly to the AST, the inner PEG nodes represent program operators, with the operands coming from the node's children. However, there are inbuilt loop operators and the PEG itself can contain loops, unlike ASTs. Therefore, PEGs are a suitable basis for loop optimising. Moreover, the correspondence

between PEGs and CFGs is such that wide changes can be made to the CFG structure by local PEG node rewrites.

6.3 Extending the Formalisation

The formalisation and framework, set out in the previous chapter, is a good foundation for considering more general graphs and their transformations. First, we need to look at how graphs can be represented, in a way that suits the functional and recursive nature of our work so far. Then the definitions and proofs can be appropriately extended. Once this is done, we can discuss how well this extension fits the kind of graph transformations that a compiler might implement.

6.3.1 Representing graphs inductively.

Trees naturally lend themselves to being inductively defined. The formalisation, in the previous chapter, and the corresponding Coq mechanisation make integral use of this feature. Graphs, on the other hand, tend to be represented as distinct sets of nodes and edges, in order to easily take account of cycles and shared nodes. This does not fit as well with the functional approach that we have taken so far.

Even though we are no longer dealing with trees, performing a depth first traversal of any graph will yield a related spanning tree. This encodes the path taken through the graph, but omits information about any edges not traversed. Hence, we will still need to represent the graph in its entirety, to avoid losing useful program information. Moreover, given the functional approach that we are taking, this representation would ideally be inductive.

Erwig's formalisation of *inductive graphs* [3] aims to address such situations and can be outlined, informally, as follows.

- Each node is represented by a unique integer, and has a label associated with it.

- A graph is then either: the empty graph, or the extension of another graph, formed by adding in a new node along with the lists of its parent and child nodes in the existing graph.

Thus, we can easily alter the definition that we had for `Tree` to an analogous definition for `Graph`.

Definition 16. A `Ctxt` is a tuple `(parents, id, label, children)`, where `id` is an integer node id, `label` is an element of some arbitrary set of graph node labels, and `parent` and `children` are corresponding lists of node `id`. We define a `Graph` as: an `Empty` graph, or a `Ctxt` added to an existing `Graph` with the operator `Amp`.

```
Graph := Empty | Amp Ctxt Graph
```

This definition cannot guarantee to yield a unique representation of a given graph. The order in which the nodes are added into a graph will determine the exact form of its graph type. For instance, consider the graph in Figure 6.1.

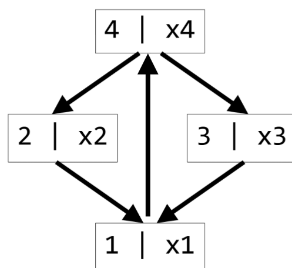

Figure 6.1: A simple graph example.

We could represent this in numerous ways, depending on the order of construction. For instance: `Amp ([1], 4, x4, [2, 3]) (Amp ([], 3, x3, [1]) (Amp ([], 2, x2, [1]) (Amp ([], 1, x1, []) Empty)))`, and: `Amp ([2, 3], 1, x1, [4]) (Amp ([4], 2, x2, []) (Amp ([4], 3, x3, []) (Amp ([], 4, x4, []) Empty)))`, both represent this graph.

Erwig proves a number of properties about this formalisation. Indeed, one motivation for defining graphs iteratively is for ease and clarity of proof. Having

defined a `gmap` function to map a function over the graph, the paper formulates and proves a fusion law, stating that mapping composed functions is equivalent to composing the mapped functions.

Definition 17. Let `f : Ctxt` $\rightarrow$ `Ctxt` be a rewrite function. Then `gmap` is defined as:

```
gmap f Empty := Empty
gmap f (Amp ctxt g) := Amp (f ctxt) (gmap f g)
```

Theorem 4. The fusion law states: `gmap f` $\circ$ `gmap f`$'$ `= gmap (f` $\circ$ `f`$'$`)`.

The fusion theorem for `gmap` is similar to the problems that we are interested in looking at, but clearly more restrictive. The functions being variously mapped and composed, in this case, are only making very local changes. They take a `Ctxt` and return a `Ctxt`, meaning that we can alter the label stored at the node itself and add or remove the edges that are stored in that `Ctxt`. The functions that we consider need to be from `Graph` to `Graph`, able to change the structure more widely, as befits an effective compiler phase.

6.3.2 Rebuilding the model.

Having found a graph definition that mirrors the inductive nature of our tree based theorems, we can now rebuild the formalisation of the previous chapter to be able to consider graphs.

Graphs, transformations and postconditions.

Given that the existing `gmap` for inductive graphs is too local for our transformation needs, we instead modify our previous `transform` definition. Rather than a necessarily postorder traversal, the graph traversal follows the inductive structure of the graph. The rewrite functions that we want to consider touch the node in question, as well as its subgraph according to the representation. Hence, the `transform` function transforms the subgraph before applying the rewrite at the node being visited.

Definition 18. We define a function `transform` that applies a given rewrite function `f : Graph` $\rightarrow$ `Graph` recursively to a given `Graph`.

```
transform f Empty := f Empty
transform f (Amp ctxt g) := f (Amp ctxt (transform f g))
```

So, the graph transformations are applied very similarly to the tree transformations, the difference being that the subgraph `g` will not necessarily contain the children of the node. Thus, the definition for applying fused rewrite functions remains as follows.

Definition 19. For two rewrite functions `f1, f2 : Graph` $\rightarrow$ `Graph` and a `Graph g`, we define `fused` as:

```
fused f1 f2 g := transform (f2 ∘ f1) g
```

where $\circ$ is standard function composition.

Moving on to the postconditions for graph transformations, we can once again decompose the graphs along the lines of the inductive representation. So, the definition of our `check` function, for checking postcondition predicates, remains similar.

Definition 20. For a predicate `p : Graph` $\rightarrow$ {`True, False`}, we define a recursive `check` function:

```
check p Empty := True
check p (Amp ctxt g) := p (Amp ctxt g) ∧ check p g
```

Next, we want to appropriately extend the definition of `ensures`, which states that a graph rewrite function ensures its corresponding postcondition if, for a graph whose subgraph satisfies the postcondition check, the result of applying the rewrite to that graph also satisfies the postcondition check.

The definition of `ensures` for `Tree`, used the list of children and trivially handled cases in which that list was empty. Here we have a single subgraph, or no subgraph if the `Graph` is `Empty`. So we must first formally define `subgraph`, and we also define an `ifexists` predicate to handle cases with no subgraph. To

do so, instead of a list, we use `option` types which will either be `Some` subgraph or `None`.

Definition 21. We define the `subgraph` of a `Graph` as follows.

```
subgraph Empty := None
subgraph (Amp ctxt g) := Some g
```

Definition 22. We also define `ifexists` to deal with `option` types. Given a predicate `p` and an `option Graph`:

```
ifexists p None := True
ifexists p (Some g) := p g
```

Definition 23. For a given rewrite function `f` and postcondition `p`, we define `ensures` as:

```
ensures f p := ∀ g, ifexists (check p) (subgraph g)
                    ⇒ check p (f g)
```

Thus, we have adapted the definitions of the `Tree` framework to apply to a broader sort of inductive graph, with minimal changes. The key choice is that the graphs are decomposed along this non-deterministic inductive structure, rather than necessarily looking at the connections within the graph itself. The next step is to examine the effects on the fusibility criteria.

Fusing pairs of transformations.

The criteria for postcondition-preserving fusion also require very few alterations. Consider `Graph` rewrite function `f1` and `f2` with corresponding postconditions `p1` and `p2` respectively. Suppose the rewrites are to fused in that order, and we wish to ensure that both postconditions are always satisfied by the result. Then we have the following criteria, which we explain informally and then define more formally.

Criterion 1. For every `Graph g` that satisfies postcondition `p1`, the result of rewriting `g` with `f2` must also satisfy `p1`. This ensures that, once `p1` has been established by `f1`, it will be preserved.

Criterion 2. For every `Graph g` whose subgraph `g′` satisfies postcondition `p2`, the subgraph of the result of rewriting `g` with `f1` must also satisfy `p2`. As, due to fusion, `g′` will have been rewritten by `f2` prior to `g` being rewritten by `f1`, this ensures that the resulting subgraph will still satisfy `p2`, as it will not be revisited to reestablish this postcondition.

Definition 24. For some rewrite function `f` and postcondition `p`, we have two criteria:

```
FC1 f p := ∀ g, check p g ⇒ check p (f g)
FC2 f p := ∀ g, ifexists (check p) (subgraph g)
            ⇒ ifexists (check p) (subgraph (f g))
```

Having adapted all of the definitions to apply to `Graph`, the resulting theorem for postcondition-proving remains essentially the same.

Theorem 5. Let `f1` and `f2` be `Graph` rewrite functions, and `p1` and `p2` be postcondition predicates. Then:

```
    ensures f1 p1 ∧ ensures f2 p2 ∧ FC1 f2 p1 ∧ FC2 f1 p2
⇒ ∀ g, check p1 (fused f1 f2 g) ∧ check p2 (fused f1 f2 g)
```

Consequently, this theorem can be proved as a corollary of two lemmas as in the previous chapter, and indeed the proofs follow the same steps, the details of which we do not reiterate here.

Lemma 13. Let `f1` and `f2` be rewrite functions and `p1` a postcondition. Then:

```
ensures f1 p1 ∧ FC1 f2 p1 ⇒ ∀ g, check p1 (fused f1 f2 g)
```

Lemma 14. Let `f1` and `f2` be rewrite functions and `p2` a postcondition. Then:

```
ensures f2 p2 ∧ FC2 f1 p2 ⇒ ∀ g, check p2 (fused f1 f2 g)
```

Fusing three or more transformations.

Given that the pairwise fusion criteria extend easily to the inductive `Graph` definition, it is clear that the criteria for fusing multiple transformations will follow suit. Thus the definitions of the criteria for postcondition-preserving fusion and

the ensuing theorem are almost unchanged from the `Tree` formulation. We will restate them here, but avoid repeating the proof details, which are in essence identical.

Definition 25. For some list of rewrite functions `fs` and postcondition `p`, we have two fusion criteria:

$\texttt{after_FC1 p fs := } \forall \texttt{ f} \in \texttt{fs, FC1 f p}$

$\texttt{before_FC2 p fs := } \forall \texttt{ f} \in \texttt{fs, FC2 f p}$

Theorem 6. Let $\texttt{fs ++ [f] ++ fs}'$ be a list of rewrite functions and `p` a postcondition. Then:

$$\texttt{ensures f p} \land \texttt{before_FC2 p fs} \land \texttt{after_FC1 p fs}'$$
$$\Rightarrow \forall \texttt{ g, check p (fused_list (fs ++ [f] ++ fs}') \texttt{ g)}$$

6.4 Illustrative Examples

To evaluate the applicability of these graph transformations, and identify the differences from the previous AST transformations, we once again consider some illustrative examples. We use the following subset of SIMPLE for this purpose.

$$s ::= s_1; s_2 \mid x := e \mid \texttt{while}\ (e)\ \{s\}$$
$$e ::= n \mid b \mid x \mid e_1 + e_2 \mid e_1 * e_2 \mid \ \ldots$$
$$b ::= \texttt{true} \mid \texttt{false}$$
$$n ::= 0 \mid 1 \mid 2 \mid \ \ldots$$

The corresponding set of node labels includes: `X := NAT Nat | BOOL Bool | VAR Var | THETA | ASG | SEQ | PLUS | MULT | ...`. The `THETA` label is taken from the nodes used in PEGs to represent while loops. The PEG has other nodes such as `eval` to encode parts of the loop such as checking the condition. But for clarity we only consider the θ node, which represents the loop induction variable.

6.4.1 Constant folding

We revisit one of the optimisations from the previous chapter, namely constant folding for arithmetic expressions.

Example 7. We define a `Graph` rewrite function `arith_fold` to implement a constant folding optimisation for arithmetic expressions.

```
arith_fold (Amp ([], i3, PLUS, [i1, i2])
           (Amp ([], i2, NAT n2, [])
           (Amp ([], i1, NAT n1, []) g')))
             = Amp ([], i3, NAT (n1 + n2), []) g')
arith_fold (Amp ([], i3, MULT, [i1, i2])
           (Amp ([], i2, NAT n2, [])
           (Amp ([], i1, NAT n1, []) g')))
             = Amp ([], i3, NAT (n1 * n2), []) g')
arith_fold g = g
```

As before, we define the postcondition to state that this transformation can make no more changes.

```
p_arith_fold (Amp ([], i3, PLUS, [i1, i2])
             (Amp ([], i2, NAT _, [])
             (Amp ([], i1, NAT _, []) _)))
               = false
p_arith_fold (Amp ([], i3, MULT, [i1, i2])
             (Amp ([], i2, NAT _, [])
             (Amp ([], i1, NAT _, []) _)))
               = false
p_arith_fold _ = true
```

We can see that this pattern matching approach to rewrites will be less effective than when considering tree transformations. Given that one graph can be represented in numerous ways, the pattern matching would have to consider

many possible cases. This makes the definitions more complicated and harder to write, but it is still possible to use a pattern matching approach, especially if the inductive graph representations are constructed in a standardised way.

6.4.2 Loop induction variable strength reduction

Consider the following generalised code snippets [88], where `k` and `c` are constant integers. For the left hand case, every time the loop body is executed, we use the result of multiplying the induction variable by the same constant `c`, before the induction variable is incremented by 1. Since multiplication is generally a more expensive operation that addition, we can optimise this program by eliminating the multiplication, to obtain the equivalent right hand case.

```
x := 0                              x := 0
i := 0                              i := 0
while (i <= k) {                    while (i <= c*k) {
    x := i * c                         x := i
    i := i + 1                         i := i + c
}                                   }
```

Figure 6.2: Loop induction variable strength reduction on pseudocode.

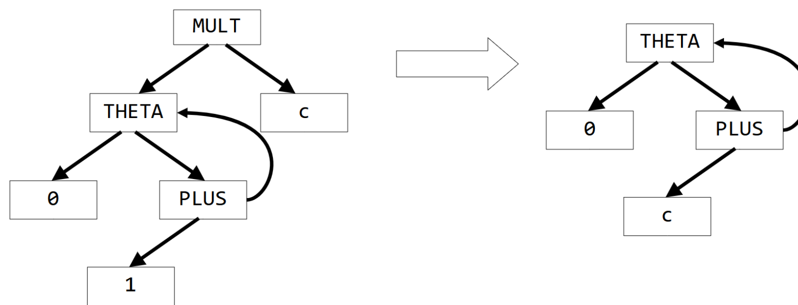

Figure 6.3: Loop induction variable strength reduction on PEGs.

This loop induction variable strength reduction (LIVSR) optimisation is illustrated on PEGs in Figure 6.3, representing a fragment of the code above. Note that θ, or `THETA`, nodes in PEGs are used to represent the induction

variables of while loops. The left child holds the start value and the right child represents how the variable is updated. Having this information encoded directly into the graph definition allows patterns like this one to be easily identified. This is also where the functionality of `Amp` nodes to encode parents is significant, otherwise it is not possible to inductively represent the cycles in our graphs.

Example 8. We define a `Graph` transformation to perform the above LIVSR.

```
livsr (Amp ([], i6, MULT, [i5, i4])
      (Amp ([], i5, NAT c, [])
      (Amp ([i3], i4, THETA, [i3, i2])
      (Amp ([], i3, PLUS, [i1])
      (Amp ([], i2, NAT 0, [])
      (Amp ([], i1, NAT 1, []) g'))))))
        = Amp ([i3], i4, THETA, [i3, i2])
          (Amp ([], i3, PLUS, [i1])
          (Amp ([], i2, NAT 0, [])
          (Amp ([], i1, NAT c, []) g')))
livsr g = g
```

Again, we see that defining transformations in this way depends on the chosen construction of the `Graph`.

6.5 Traversal Order & Inductive Representation

As we have demonstrated, the tree transformation framework adapts naturally to an inductive notion of graphs. This inductive definition is the key that allows the extension to easily propagate through. Therefore, it is important to establish that the definition makes sense. The criteria and theorems are meaningless if they do not accurately reflect the situation at hand.

A significant observation is that, in this formalisation, the graph traversal order is determined by the order in which the inductive representation is constructed. The subgraph of a given graph represented this way is entirely

dependent on the construction, and may not necessarily contain any of the children of the node in question. This does not affect the soundness of such an approach, but may limit its usefulness.

However, if we make sure to use a depth-first order to construct the inductive representation, then we will essentially create a spanning tree which encodes the information of all graph edges. The order in which nodes are added to the graph dictates the order in which the nodes are visited by transformations. This means that the traversal order can be enforced by conditions on the graph structure itself.

Moreover, the graphs that we deal with in a compiler are constructed by some part of the compiler pipeline itself, often based on an intermediate language. This means that we already know, to some extent, what the graph structure is likely to look like. An interesting avenue of further research would be to examine how best to construct these inductive graph definitions to work effectively in a compiler.

6.6 Discussion

This chapter has adapted the work of the previous chapter to consider a more general form of graph. Although tree structures are commonly found in the frontend of compilers, many optimisations further down the pipeline rely on information that is more easily encoded in graphs with shared nodes and loops. To extend from trees to graph transformation fusion, we have selected an inductive graph definition. This makes the transition as smooth as possible.

As we did for tree transformation fusion, we have looked at simple compiler transformations in order to evaluate the actual usefulness of the definitions and criteria specified. These examples show that is possible to define realistic transformations in this way, however it is not always easy to make them effective. In particular, although the inductive definition of graphs makes the specifications and proofs nicer, it does make transformations based on pattern matching less useful, since the representation of a graph could change depending on the order

in which the nodes are added.

In addition to this, the traversal order for transformations is also dependent on the order of nodes in the inductive representation. As such, we cannot simply assume that the subgraph of a node contains all or, indeed, any of its children. This means that the traversal is determined by the graph rather than the transformations or the framework at large. Hence, to make transformations properly useful we would need to specify conditions on the graph construction process, essentially imposing a consistent traversal order.

Chapter 7

Discussion & Conclusions

7.1 Thesis Contributions

The Methodology chapter set out the research questions that we wanted to explore in this thesis. Now we can revisit these questions and examine the extent to which our work has addressed them.

First, to define the scope of the work in this thesis, we specified the kind of compiler phases to consider. We defined the tree and graph structures, and corresponding transformations, that we were interested in. Due to taking the miniphase framework [1] as a basis, and since they are so ubiquitous, we started with abstract syntax trees (ASTs) as our intermediate representation. We then moved on to more general graph structures, taking inductive graphs [3] and program expression graphs (PEGs) [23] as inspiration.

What does it mean for compiler phases to be fusible?

Having looked at different ideas of fusibility from similar related work, we settled on postcondition-preservation, such as is used for miniphases, as a useful approximation. Postconditions that are defined by the compiler developer can encompass the necessary outcome of a given compiler phase. This tells us which parts of the transformation need to be preserved by the fusion process, and al-

lows us to widely permit fusion as long as it does preserve them.

This definition is better suited to fusing optimisations than approaches that strictly require fused and unfused transformations to produce identical results. Optimisations are often more effective when working together and this can be one of the benefits of fusion. However, postconditions cannot always precisely capture the nuances of optimisation. Hence, there may be other qualities that could be considered to augment the definition of fusibility, along with postcondition preservation.

Can we determine whether phases are fusible without attempting to combine them?

We have developed criteria, namely `FC1` and `FC2` as specified in Section 5.3 Definition 7, for assessing fusibility before phases have been fused. This is key to being able to reason modularly about modular compiler phases. Each criterion considers only one transformation and one postcondition, making them simpler and easier to check, as well as more robust against local implementation changes within the compiler. Moreover, since the criteria are parameterised over the defined postconditions, it may be possible to weaken the postcondition in order to permit fusion.

We have also verified that these criteria are sound, so that any transformations which satisfies them will always preserve the postconditions in question. This formal underpinning gives us strong guarantees. It is worth noting, however, that there is no reason to assume that the criteria are complete. So, there could be cases that are fusible but that the criteria do not pick up.

After defining an appropriate idea of fusibility and deriving modular criteria for reasoning about this, we assessed the feasibility of the work by looking at several simplistic examples. From this we could determine that the modular approach had potential. We could apply it to simple cases of transformations found in realistic compiler phases.

However, there is much more scope for attempting to implement a more

complex real-world compiler, and analysing the outcome. This would provide further indication as to whether the modular reasoning approach is usable or useful. Furthermore, in trying to implement examples, we found that some design choices that facilitated the specification and proof process, made the application process more difficult. Hence, such findings could be used to refine the framework further, in terms of developer experience.

7.2 Using Modular Criteria

The main reason behind developing a set of formal criteria for postcondition-preserving fusion is to allow modular reasoning. We can think about `f1` and `f2` individually, rather than having to determine whether `fused f1 f2` will always establish the relevant postconditions. Namely, for pairwise fusion, we are checking:

`ensures f1 p1`, `ensures f2 p2`, `FC1 f2 p1`, and `FC2 f1 p2`,

rather than:

$\forall$`t, check p1 (fused f1 f2 t)` $\wedge$ `check p2 (fused f1 f2 t)`.

This imparts the advantages of modularity. For instance, a developer working on a specific transformation, does not need to understand the implementation of other transformations, in order to assess fusibility. Instead, they only need the postconditions of those transformations. They can then check these postconditions and their own transformation against the relevant fusibility criterion. The postconditions and the assurances of their suitability form an interface between the transformations that allows an appropriate level of information hiding.

Moreover, the modularity allows components to be changed without affecting the entire system. This is particularly relevant when a large number of

transformations is being fused. Without modularity, the entire fused series of transformations needs to be reevaluated for postcondition-preservation, including transformations and postconditions that remain the same. With modular criteria, only the relationships with changed components need to be reestablished. So only a fraction of the checks need to be rechecked, while the rest remain valid.

In this section, we briefly explore a range of techniques that could be used to check the criteria prior to fusion. Each of these approaches has its benefits and may be suited to different situations. Modularity is beneficial in each of these cases.

7.2.1 Using a proof assistant.

Proof assistants, or interactive theorem provers, allow developers to implement software and formally prove that it satisfies desired properties, such as our fusion criteria. Having simple modular tree transformations to reason about will make this process easier for the inexperienced user, as there is generally a steep learning curve in using proof assistants. Moreover, if the proofs are reasonably simple, there is more chance that the small amounts of proof automation, present in such tools, will be helpful.

The Coq proof assistant [41] is a popular choice for verifying software. We have used Coq to mechanise and check the formalisation and proofs in this paper. This not only substantiates the work, but also leaves behind a framework that could be used as a template to implement specific tree transformations and prove that the fusion criteria hold.

Verified code implemented in Coq can be directly extracted into a number of languages, such as OCaml and Haskell. Hence, the software does not need to be implemented a second time for verification. There are also projects like `hs-to-coq` [89, 90], which aims to automatically translate Haskell code into Gallina, the Coq specification language. Such tools help to bridge the gap between verification, via theorem proving, and software implementation.

7.2.2 Automated static analysis.

Another, more automated, possibility is to use some form of static code analysis software. Static analysis examines the behaviour of a program without executing it, and can exploit tools like SMT solvers to check that given properties hold. Such approaches tend to be less expressive than using a proof assistant, but provide the benefit of automation.

With automated verification methods, modularity is valuable. Having smaller components and simpler properties to check helps to avoid issues such as state-explosion. Moreover, many such methods search for a counterexample input, if a proof cannot be found. A counterexample is most useful if it can be directly linked with a specific section of code, to effectively diagnose the problem. This is more likely to happen if the verification effort is highly modular.

7.2.3 Runtime checks.

Contract syntax is often used to express properties to be checked at runtime. This can exist in native form in a language, such as JML, or as part of a library, such as `Predef` in Scala. Keywords like `require` and `ensuring` are used to express preconditions and postconditions at a given point in code. Exceptions may be raised if these conditions are not met. Contracts can be permanently in place during runtime or toggled for use only during testing.

If we are implementing modular tree transformations, it is necessary for the conditions that we are checking to also be modular. It is not straightforward to express, using contracts, a property referring to multiple tree transformations which are defined in different parts of code. Therefore, our criteria are useful here, in that each only relates to one tree transformation. Postconditions can then be expressed as part of the contract syntax.

7.2.4 Property-based testing.

Property-based testing frameworks are designed to test for specific properties, using a large number of appropriate random inputs. This can be used in a

similar way as contract syntax, during testing, however it removes the required properties from the code itself. It also allows the user to express properties about multiple elements, rather than just one point in the code.

The advantage of modularity here, is that the tests can be very specific. If a given test fails, it is easy to localise which criteria was not met, as well as which particular tree transformation caused the problem.

7.3 Potential Future Work

The work in this thesis has suggested an approach for achieving modular, efficient, and correct program transformations, via formal and modular reasoning techniques. We have also proved that these techniques are sound and demonstrated with simple examples that they could be useful. However, there are many areas that this thesis does not touch on, and that could form the basis of interesting future work.

7.3.1 Realistic compiler implementation.

Whilst the examples that illustrate this work are transformations that would commonly be found in a realistic compiler, they only represent a small fraction of the work that a compiler necessarily does. Therefore, a good method of evaluating the usability of the modular reasoning approach that we have developed would be to use it in implementing a more substantial example. Given that the inspiration for these ideas was the Dotty miniphase compiler [1], that would be an obvious place to start, especially since the source code is readily available. Moreover, taking the JVM as a target provides a good starting point for looking at compiling any other languages that typically choose to run on JVM. A realistic application of the framework would then facilitate other research, as we detail below.

7.3.2 Evaluating modularity gains.

Throughout this thesis, we have taken modularity to be an imprecise and unmeasured quantity, validated by anecdotal evidence. However, there are frameworks and techniques for quantifying the modularity of a piece of software and the value that that modularity adds. For instance, design structure matrices can model modular designs and evaluate the benefits [6, 91]. In such approaches, real options techniques can be used to consider the added value of being able to easily replace components if better implementations are discovered.

7.3.3 Evaluating performance gains.

We have not, thus far, directly discussed the performance impact of fusing compiler phases, instead relying on positive results from other work such as the Dotty compiler [1]. Therefore, another implementation of this kind of compiler would provide the opportunity for further rigorous benchmarking of performance gains. In addition to this, if the criteria for fusibility are checked with any degree of automation, it would be useful to compare the time spent on this compared to checking less modular criteria.

7.3.4 Sophisticated compiler correctness.

The approach to compiler correctness in this work has been somewhat naive and simplistic compared to the plethora of work focused on verifying real-world compilers. Hence, there is plenty of room to explore how the fusion techniques and modular reasoning criteria with respect to existing compiler correctness projects. In particular, many successfully verified compilers, such as CompCert [27], are directly extracted from their proofs of correctness. As modularity is already key in verification efforts, if a compiler is to be realistically used then performance also needs to be taken into account. Thus, it may be that fusion is of benefit in such situations.

7.4 Thesis Conclusions

Ordinarily the increased number of compiler phases promoted by modularity would result in an increased number of graph traversals and, hence, worse performance. Automatically fusing graph transformations can avoid this trade-off of modularity against performance. Fusion allows multiple transformations to be performed in a single traversal, mitigating the performance impact. Such a strategy has been successfully adopted for AST transformations as miniphases in the Dotty Scala compiler, implemented by Petrashko et al. [1].

A crucial consideration, when fusing compiler phases, is correctness. Fused phases must continue to be useful, by performing their intended task. Most work on fusing tree traversals or transformations deems fusion successful if the fused transformations will produce an identical outcome to the same separate transformations run consecutively. However, this precludes some fusion opportunities that would still be beneficial. Moreover, many related techniques focus on a highly restricted set of transformations, in order to prove such soundness guarantees.

Instead, we have argued for a broader notion of what we allow to be fusible, namely postcondition-preserving fusion. We use postconditions to encode the required behaviours of a graph transformation, allowing the developer of the transformation to specify what particular behaviour is important to them. We can then reason about whether that behaviour is preserved, rather than trying to preserve behaviour that is merely coincidental.

We have also derived and verified criteria that are sufficient to guarantee that a given set of graph transformations can be successfully fused, with respect to a given set of postconditions. Instead of reasoning about the final fused transformation, we are able to reason about the transformations individually. This will allow modular verification or testing which appropriately complements the modularity of the implemented compiler phases.

Bibliography

[1] Dmitry Petrashko, Ondřej Lhoták, and Martin Odersky. Miniphases: compilation using modular and efficient tree transformations. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 201–216. ACM, June 2017.

[2] Dmytro Petrashko. *Design and implementation of an optimizing type-centric compiler for a high-level language.* PhD Thesis, EPFL, 2017.

[3] Martin Erwig. Inductive graphs and functional graph algorithms. *Journal of Functional Programming*, 11(5):467–492, 2001.

[4] David Lorge. Parnas. A technique for software module specification with examples. *Communications of the ACM*, 15(5):330–336, May 1972.

[5] David Lorge. Parnas, Paul C. Clements, and David M. Weiss. The modular structure of complex systems. *IEEE Transactions on software Engineering*, 3:259–266, March 1985.

[6] Carliss Young Baldwin and Kim B. Clark. *Design rules: The power of modularity*, volume 1. MIT Press, 2000.

[7] Donald E. Knuth. Invited papers: History of writing compilers. In *Proceedings of the 1962 ACM National Conference on Digest of Technical Papers*, page 43, September 1962.

[8] Niklaus Wirth. The programming language Pascal. *Acta Informatica*, 1(1):35–63, March 1971.

[9] Niklaus Wirth. The design of a PASCAL compiler. *Software: Practice and Experience*, 1(4):309–333, October 1971.

[10] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization*, pages 75–86. IEEE, March 2004.

[11] Dipanwita Sarkar, Oscar Waddell, and R. Kent Dybvig. A nanopass infrastructure for compiler education. *ACM SIGPLAN Notices*, 39(9):201–212, September 2004.

[12] Andrew W. Keep and R. Kent Dybvig. A nanopass framework for commercial compiler development. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, pages 343–350, September 2013.

[13] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the scala programming language. Technical report, EPFL, 2004.

[14] Alexander Aiken. Cool: A portable project for teaching compiler construction. *ACM Sigplan Notices*, 31(7):19–24, July 1996.

[15] Oege de Moor, Simon Peyton Jones, and Eric Van Wyk. Aspect-oriented compilers. In *International Symposium on Generative and Component-Based Software Engineering*, pages 121–133, September 1999.

[16] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In *European Conference on Object-Oriented Programming*, pages 406–431, July 1993.

[17] Rhys Weatherly. Treecc: An aspect-oriented approach to writing compilers. *Free Software Magazine*, 1(2), February 2002.

[18] Görel Hedin and Eva Magnusson. JastAdd — an aspect-oriented compiler construction system. *Science of Computer Programming*, 47(1):37–58, April 2003.

[19] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *European Conference on Object-Oriented Programming*, pages 327–354, June 2001.

[20] Xiaoqing Wu, Suman Roychoudhury, Barrett R. Bryant, Jeffrey G. Gray, and Marjan Mernik. A two-dimensional separation of concerns for compiler construction. In *Proceedings of the 2005 ACM Symposium on Applied Computing*, pages 1365–1369, March 2005.

[21] James Stanier and Des Watson. Intermediate representations in imperative compilers: A survey. *ACM Computing Surveys (CSUR)*, 45(3):1–27, July 2013.

[22] Frances E. Allen. Control flow analysis. *ACM SIGPLAN Notices*, 5(7):1–19, July 1970.

[23] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: A new approach to optimization. *Logical Methods in Computer Science*, 7(1), 2011.

[24] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Translating between PEGs and CFGs. Technical report, Department of Computer Science and Engineering, UC San Diego, 2008.

[25] John McCarthy and James Painter. Correctness of a compiler for arithmetic expressions. *Mathematical Aspects of Computer Science: Proceedings of Symposia in Applied Mathematics*, 19, December 1967.

[26] Maulik A. Dave. Compiler verification: A bibliography. *ACM SIGSOFT Software Engineering Notes*, 28(6):2, November 2003.

[27] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, July 2009.

[28] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 283–294, June 2011.

[29] Robin Milner and Richard Weyhrauch. Proving compiler correctness in a mechanized logic. *Machine Intelligence*, 7:51–70, 1972.

[30] Jianzhou Zhao, Santosh Nagarakatte, Milo MK Martin, and Steve Zdancewic. Formalizing the LLVM intermediate representation for verified program transformations. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 427–440, January 2012.

[31] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: a verified implementation of ML. *ACM SIGPLAN Notices*, 49(1):179–191, January 2014.

[32] William R. Bevier, Warren A. Hunt, J. Strother Moore, and William D. Young. An approach to systems verification. *Journal of Automated Reasoning*, 5(4):411–428, December 1989.

[33] William D. Young. Verified compilation in micro-Gypsy. *ACM SIGSOFT Software Engineering Notes*, 14(8):20–26, November 1989.

[34] J. Strother Moore. A mechanically verified language implementation. *Journal of Automated Reasoning*, 5(4):461–492, December 1989.

[35] Wolfgang Goerigk. Proving Preservation of Partial Correctness with ACL2: A Mechanical Compiler Source Level Correctness Proof. In *Proceedings of the ACL2 2000 Workshop*, October 2000.

[36] Wolfgang Goerigk, Axel Dold, Thilo Gaul, Gerhard Goos, Andreas Heberle, Friedrich W. Von Henke, Ulrich Hoffmann, Hans Langmaack, Holger Pfeifer, Harald Ruess, and Wolf Zimmermann. Compiler correctness and implementation verification: The verifix approach., 1996.

[37] John Hannan and Frank Pfenning. Compiler verification in LF. In *Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 407–418. IEEE, June 1992.

[38] Gerwin Klein and Tobias Nipkow. Verified bytecode verifiers. *Theoretical Computer Science*, 298(3):583–626, April 2003.

[39] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 207–220. ACM, October 2009.

[40] Ramana Kumar. *Self-compilation and self-verification.* PhD Thesis, University of Cambridge, Computer Laboratory, 2016.

[41] The Coq Development Team. The Coq Proof Assistant., Version 8.7.0.

[42] Adam Chlipala. A certified type-preserving compiler from lambda calculus to assembly language. *ACM SIGPLAN Notices*, 42(6):54–65, June 2007.

[43] Adam Chlipala. A verified compiler for an impure functional language. *ACM Sigplan Notices*, 45(1):93–106, January 2010.

[44] Adam Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant.* MIT Press, 2013.

[45] Andreas Lochbihler. Verifying a compiler for Java threads. *ESOP*, 6012:427–447, March 2010.

[46] Jeehoon Kang, Yoonseung Kim, Chung-Kil Hur, Derek Dreyer, and Viktor Vafeiadis. Lightweight verification of separate compilation. *ACM SIGPLAN Notices*, 51(1):178–190, January 2016.

[47] Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel. Compositional CompCert. *ACM SIGPLAN Notices*, 50(1):275–287, May 2015.

[48] Daniel Patterson and Amal Ahmed. The next 700 compiler correctness theorems (functional pearl). *In Proceedings of the ACM on Programming Languages*, ICFP(3):1–29, July 2019.

[49] Marco Patrignani, Amal Ahmed, and Dave Clarke. Formal approaches to secure compilation: A survey of fully abstract compilation and related work. *ACM Computing Surveys*, 51(6):1–36, February 2019.

[50] Dennis Volpano and Geoffrey Smith. A type-based approach to program security. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 607–621. Springer, April 1997.

[51] David Tarditi, Greg Morrisett, Perry Cheng, Chris Stone, Robert Harper, and Peter Lee. TIL: A type-directed optimizing compiler for ML. *ACM Sigplan Notices*, 31(5):181–192, May 1996.

[52] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(3):527–568, May 1999.

[53] Kathi Fisler and Shriram Krishnamurthi. Modular verification of collaboration-based software designs. In *Proceedings of the 8th European Software Engineering Conference*, pages 152–163, September 2001.

[54] Harry C. Li, Kathi Fisler, and Shriram Krishnamurthi. The influence of software module systems on modular verification. In *International SPIN Workshop on Model Checking of Software*, pages 60–78, April 2002.

[55] Charles A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[56] Bertrand Meyer. Eiffel: A language and environment for software engineering. *Journal of Systems and Software*, 8(3):199–246, June 1988.

[57] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, October 1992.

[58] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, May 2006.

[59] Martin Odersky. Contracts for Scala. In *International Conference on Runtime Verification*, pages 51–57. Springer, November 2010.

[60] Cliff B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(4):596–619, October 1983.

[61] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. *ACM SIGPLAN Notices*, 46(4):53–64, May 2011.

[62] Cheon Yoonsik and Gary T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In *European Conference on Object-Oriented Programming*, pages 231–255. Springer, June 2002.

[63] Régis Blanc, Viktor Kuncak, Etienne Kneuss, and Philippe Suter. An overview of the Leon verification system: Verification by translation to recursive functions. In *Proceedings of the 4th Workshop on Scala*, pages 1–10, July 2013.

[64] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, March 2008.

[65] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *International Conference on Computer Aided Verification*, pages 171–177. Springer, July 2011.

[66] Sagar Chaki, Edmund M. Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in C. *IEEE Transactions on Software Engineering*, 30(6):388–402, August 2004.

[67] Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric, and Zachary Tatlock. QED at large: A survey of engineering of formally verified software. *Foundations and Trends in Programming Languages*, 5(2-3):102–281, September 2019.

[68] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic.* Springer Science & Business Media, 2002.

[69] Lars Hupel and Viktor Kuncak. Translating Scala programs to Isabelle/HOL. In *International Joint Conference on Automated Reasoning*, pages 568–577. Springer, June 2016.

[70] Hannes Mehnert. Kopitiam: Modular incremental interactive full functional static verification of Java code. In *NASA Formal Methods Symposium*, pages 518–524. Springer, April 2011.

[71] Youngjoon Jo and Milind Kulkarni. Enhancing locality for recursive traversals of recursive structures. *ACM SIGPLAN Notices*, 46(10):463–482, October 2011.

[72] Youngjoon Jo and Milind Kulkarni. Automatically enhancing locality for tree traversals with traversal splicing. *ACM SIGPLAN Notices*, 47(10):355–374, October 2012.

[73] Yusheng Weijiang, Shruthi Balakrishna, Jianqiao Liu, and Milind Kulkarni. Tree dependence analysis. *ACM SIGPLAN Notices*, 50(6):314–325, June 2015.

[74] Samyam Rajbhandari, Jinsung Kim, Sriram Krishnamoorthy, Louis-Noël Pouchet, Fabrice Rastello, Robert J. Harrison, and Ponnuswamy Sadayappan. A domain-specific compiler for a parallel multiresolution adaptive numerical simulation environment. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 40. IEEE Press, November 2016.

[75] Samyam Rajbhandari, Jinsung Kim, Sriram Krishnamoorthy, Louis-Noël Pouchet, Fabrice Rastello, Robert J. Harrison, and Ponnuswamy Sadayappan. On fusing recursive traversals of k-d trees. In *Proceedings of the 25th International Conference on Compiler Construction*, pages 152–162. ACM, March 2016.

[76] Laith Sakka, Kirshanthan Sundararajah, and Milind Kulkarni. TreeFuser: a framework for analyzing and fusing general recursive tree traversals. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):76, October 2017.

[77] Xiaokang Qiu and Yanjun Wang. A decidable logic for tree data-structures with measurements. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, Lecture Notes in Computer Science, Vol. 11388, pages 318–341. Springer, January 2019.

[78] Philip Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, 1990.

[79] Wei-Ngan Chin. Safe fusion of functional expressions. *ACM SIGPLAN Lisp Pointers*, V(1):11–20, January 1992.

[80] Andrew Gill, John Launchbury, and Simon L. Peyton Jones. A short cut to deforestation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 223–232. ACM, July 1993.

[81] Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream fusion: From lists to streams to nothing at all. *ACM SIGPLAN Notices*, 42(9):315–326, October 2007.

[82] Geoffrey Mainland, Roman Leshchinskiy, and Simon L. Peyton Jones. Exploiting vector instructions with generalized stream fusion. *Communications of the ACM*, 60(5):83–91, April 2017.

[83] Akihiko Takano and Erik Meijer. Shortcut deforestation in calculational form. In *Conference on Functional Programming Languages and Computer Architecture*, pages 306–313. ACM, October 1995.

[84] Erik Meijer, Maartan Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Conference on Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science, Vol. 523, pages 124–144. Springer, August 1991.

[85] Michael Benjamin Stepp. *Equality saturation: engineering challenges and applications.* PhD Thesis, UC San Diego, 2011.

[86] Ross Tate. *Equality saturation: using equational reasoning to optimize imperative functions.* PhD Thesis, UC San Diego, 2012.

[87] Eleanor Davies, 2021. https://github.com/EleanorRD/PhDThesis.

[88] Ross Tate, Michael Stepp, and Sorin Lerner. Generating compiler optimizations from proofs. *ACM SIGPLAN Notices*, 45(1):389–402, January 2010.

[89] Antal Spector-Zabusky, Joachim Breitner, Christine Rizkallah, and Stephanie Weirich. Total Haskell is reasonable Coq. In *Proceedings of*

the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, pages 14–27. ACM, January 2018.

[90] Joachim Breitner, Antal Spector-Zabusky, Yao Li, Christine Rizkallah, John Wiegley, and Stephanie Weirich. Ready, set, verify! Applying hs-to-coq to real-world Haskell code (Experience report). *Proceedings of the ACM on Programming Languages*, 2(ICFP):89, July 2018.

[91] Kevin J. Sullivan, William G. Griswold, Yuanfang Cai, and Ben Hallen. The structure and value of modularity in software design. *ACM SIGSOFT Software Engineering Notes*, 26(5):99–108, September 2001.