Formal verification of translation validators

Jean-Baptiste Tristan

▶ To cite this version:

Jean-Baptiste Tristan. Formal verification of translation validators. Génie logiciel [cs.SE]. Université Paris-Diderot - Paris VII, 2009. Français. tel-00437582

HAL Id: tel-00437582

https://tel.archives-ouvertes.fr/tel-00437582

Submitted on 30 Nov 2009

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

PARIS. DIDEROT UNIVERSITY (Paris 7)

GRADUATE SCHOOL OF MATHEMATICAL SCIENCE OF PARIS

Ph.D.

Computer Science

JEAN-BAPTISTE TRISTAN

FORMAL VERIFICATION OF TRANSLATION VALIDATORS

Advisor: Xavier LEROY

November 6, 2009

JURY

M. Albert Cohen	, examiner
M. Patrick Cousot	, examiner
M. Roberto Di Cosmo	, examiner
M. Nicolas Halbwachs	, examiner
M. Thomas Jensen	, reviewer
M. Xavier Leroy	, Ph.D. advisor
M. George Necula	, reviewer

To Wŭ Wēi

Acknowledgments

I owe so much to my Ph.D. advisor, Xavier Leroy. Thanks to him, my three years of Ph.D. have been a very useful and enjoyable time. I greatly benefited from his wonderful scientific judgment, his ability to communicate science in a precise, clear and meaningful way and his hacking skills. Moreover, he has been a great adviser. He managed to give me the freedom of pursuing my goals and interests while invariably being there to provide guidance. But Xavier is more than a great scientist. It has always been such a pleasure to listen to him when he speaks about literature or classical music. He really is a role model. Thank you so much Xavier.

I would like to thank the members of my Ph.D. committee, in particular Thomas Jensen and George Necula who agreed on reviewing my dissertation and gave interesting comments.

I would also like to thank my friends, colleagues or mentors. Benoît Razet who has read and commented on every bit of this dissertation, and who has listened to every idea I had during these three years. Paul Govereau, Kelly Heffner, and Kevin Redwine, who have convinced me that doing research can be great, who made my time in Cambridge Massachusetts so enjoyable, and who make it possible for me to live this once again. Finally, I would like to thank all the members of the Gallium and Moscova groups at INRIA, in particular Gérard Huet, Damien Doligez, Jean-Jacques Lévy, and Arthur Charguéraud.

Last but not least, I would like to thank my family, especially my parents, Martine & Bernard Tristan who have always been there for me, and finally my wife, Wēi, who has so greatly contributed to my thesis by helping me making the right decisions everytime I was facing a decisive choice. Only I know to what extent doing this Ph.D. would never have been possible without her.

Contents

1 Introduction **11**

1.1 Trustworthy compilation . . . 11

1.2 Formal verification of translation validators . . . 14

1.3 Case studies: formally verified validation of 4 optimizations . . . 17

1.4 A brief history of translation validation . . . 20

1.5 Notes about this document . . . 24

2 Setting **25**

2.1 The Compcert verified compiler . . . 25

2.2 Elements of syntax and semantics . . . 27

2.2.1 Programs . . . 27

2.2.2 Values and memory states . . . 28

2.2.3 Global environments . . . 29

2.2.4 Traces . . . 30

2.2.5 Operators, conditions and addressing modes . . . 30

2.3 The RTL intermediate language . . . 31

2.3.1 Syntax . . . 32

2.3.2 Semantics . . . 32

2.4 The Mach intermediate language . . . 35

2.4.1 Syntax . . . 36

2.4.2 Semantics . . . 36

2.5 Experimental protocol . . . 39

3 List scheduling **41**

3.1 List scheduling . . . 41

3.2 Formalization of symbolic evaluation . . . 42

3.2.1 Symbolic expressions . . . 44

3.2.2 Algorithm for symbolic evaluation . . . 46

3.2.3 Properties of symbolic evaluation . . . 47

3.3 A validator for list scheduling . . . 48
3.3.1 Validation at the level of blocks . . . 48
3.3.2 Validation at the level of function bodies . . . 49
3.4 Proof of correctness . . . 49
3.4.1 Correctness of block validation . . . 49
3.4.2 Correctness of function bodies validation . . . 50
3.5 Discussion . . . 51
3.5.1 Implementation . . . 51
3.5.2 Experimental evaluation and Complexity analysis . . . 51
3.5.3 A note on diverging executions . . . 53
3.5.4 Conclusion . . . 53

4 Trace scheduling 55
4.1 Trace scheduling . . . 55
4.2 A tree-based representation of control and its semantics . . . 57
4.2.1 Conversion to tree-based representation . . . 60
4.2.2 Correctness and completeness of the conversion . . . 63
4.3 A validator for trace scheduling . . . 63
4.3.1 Validation at the level of trees . . . 63
4.3.2 Validation at the level of function bodies . . . 64
4.4 Proof of correctness . . . 65
4.4.1 Correctness of the validation over trees . . . 65
4.4.2 Correctness of validation over function bodies . . . 65
4.5 Discussion . . . 65
4.5.1 Implementation . . . 66
4.5.2 Experimental evaluation and complexity analysis . . . 66
4.5.3 Conclusion . . . 68

5 Lazy code motion 69
5.1 Lazy code motion . . . 69
5.2 A validator for lazy code motion . . . 71
5.2.1 General structure . . . 71
5.2.2 Verification of the equivalence of single instructions . . . 72
5.2.3 Verifying the flow of control . . . 74
5.3 Proof of correctness . . . 78
5.3.1 Simulating executions . . . 78
5.3.2 The invariant of semantic preservation . . . 79
5.3.3 A little bit of proof design . . . 81

5.3.4 Verification of the equivalence of single instructions 82
5.3.5 Anticipability checking 83
5.4 Discussion 84
5.4.1 Implementation 84
5.4.2 Experimental evaluation and complexity analysis 84
5.4.3 Completeness 85
5.4.4 Reusing the development 86
5.4.5 Conclusion 87

6 Software pipelining 89
6.1 Software pipelining 89
6.1.1 Effects of software pipelining 92
6.2 Overview of the design 94
6.2.1 A sound symbolic model 94
6.2.2 Satisfiability of the model 96
6.3 Symbolic evaluation modulo observables 97
6.4 Reasoning over symbolic evaluations 101
6.4.1 Decomposition 101
6.4.2 Rewriting of symbolic states 103
6.5 A validator for the software pipeliner 104
6.5.1 Finite characterizations 104
6.5.2 The validator 106
6.6 Soundness of the symbolic model 107
6.6.1 Separation of denotation and control 108
6.6.2 A symbolic model of the loops 109
6.7 Discussion 111
6.7.1 Implementation and preliminary experiments 111
6.7.2 Related work 113
6.7.3 Conclusion 113

7 Conclusion 115
7.1 Summary of the contributions 115
7.2 Future work 116
7.3 Assessment 117

Chapter 1

Introduction

In recent years, formal methods have attracted considerable interest and met some success in the safety-critical software industry, particularly in avionics [Jac09, Har08]. The verification tools that are used to catch bugs and enforce safety policies are generally applied on the source code of a critical software; but what we want, in the end, is that the *executable* code is free of bugs and satisfies its safety policy. Yet, verifying the source code has its advantages: the design of verification tools can be simpler when dealing with a high-level language with clear restrictions and abstractions than when dealing with machine-level, unstructured code. In this context, the compiler has to be trusted not to introduce bugs of its own in the executable code it generates from the verified source code.

A problem is that compilers, and especially optimizing compilers, are complex pieces of software that perform subtle transformations over the programs being compiled, exploiting the results of delicate static analyses. Moreover, compilers are difficult to test thoroughly. Consequently, compilers are sometimes incorrect, which may result at best in a crash during compilation and at worse in the silent generation of a bad executable from a correct source program. The latter case is especially alarming: the compiler can potentially invalidate the guarantees established by applying formal methods to the source code.

The standard approach to weeding out compilation bugs in the safety-critical industry is heavy testing of the compiler and manual inspection of the generated code. This makes compiler development costly and usually requires to turn off optimizations. An alternative to make the compiler more trustworthy is to apply formal methods on the compiler itself to verify its correctness.

1.1 Trustworthy compilation

Verifying the correctness of compilers is hardly a new problem. In a paper published in 1963 [McC63], John McCarthy refers to this problem as "*one of the most interesting and useful goals*

for the mathematical science of computation". In another paper published in 1967 [MP67], he gives a paper-and-pencil proof of a compiler for arithmetic expressions. Since then, many proofs for parts of compilers have been published. Dave's paper [Dav03] is a comprehensive bibliography of those works. We restrict our attention to the methods that allow to verify that the implementation of the compiler is correct, as opposed to verifying an abstract model of the compiler or the algorithms that it uses.

A different approach, *certifying compilation* [Nec97], consists in generating, along with the object code, concrete evidence that the object code satisfies some safety properties. This approach is generally used to attest that the object code satisfy a specific property, in this dissertation, we set out to make sure that the compiler preserves all the properties of the source program.

There are two approaches that, in practice, allow to make the compiler more trustworthy: formal compiler verification and translation validation.

Formal compiler verification Formal compiler verification applies mechanized proof techniques to the compiler in order to prove, once and for all, that the generated code is semantically equivalent to the source code and, therefore, enjoys all the safety properties that were established by source-level verification. The seminal work in the formal verification of compilers is the one by Milner and Weyrauch [MW72] in 1972, for the compilation of arithmetic expressions, and the first realistic formal verification of a compiler is due to Moore [Moo89] in 1989, for an assembly-like source language.

From the bird's eye, the formal verification of software consists in:

- Specifying what the program is intended to do, using a specification language that has precisely-defined semantics based on mathematical logic.
- Providing a comprehensive proof, based on well-established mathematical reasoning and verifiable by a machine, that the software itself (not a model of it) satisfies its specification.

The formal verification of a compiler fits this general pattern. The high-level specification of the compiler is a semantic preservation property: whatever machine code it generates must execute exactly as prescribed by the semantics of the source program. Note that the specification of the compiler is expressed in terms of mathematical semantics for its source and target language. There are several tools that can be used to develop such software. Some tools start with industry-standard programming languages (such as C, Java or C#), extend them with a specification language, and provide a verification condition generator along with theorem provers to discharge the generated proof obligations. Representative examples of this approach are ESC/Java, Spec#, and Caduceus/Frama-C. Another approach is to use proof assistants [Geu09, Wie08]: computer systems that allow to do mathematics on a computer and develop verified software. This approach is heavier to use than other automatic formal methods but it

enables proving richer functional specifications requiring proof invariants that are stronger than the specification itself and cannot be found by a machine.

More generally, formal verification is of great interest in the development of safety-critical software because it provides strong safety guarantees [Hal08, Gon08]. Critical computer systems are usually designed so as to minimize the number of components that the safety of the final software depends on. Those components constitute what is called the *trusted computing base* (TCB) because there is no choice but to trust that these components are correct. Consider as an example a piece of software on which we apply a model checking algorithm to ensure that the software does exactly what it is intended to do. In such a case, the TCB contains the model checker itself: if it is not correctly enforcing the functional properties, the software may be faulty. As another example, if we formally verify that a software satisfies some specification, the TCB contains the specification. In the end, safety always depend on some amount of software or specification that have to be trusted. From this standpoint, formal verification is interesting because a specification expressed formally in the universal language of mathematics is less subject to mis-interpretation than an on-paper specifications or a piece of code written in a computer language with a brittle semantics.

Computer systems, unlike other engineering artifacts, have a discontinuous behavior. In engineering, behaviors are generally continuous: systems can be designed so that, within well-known conditions, a small change in the environment of a system leads to a small and local change of the behavior of the system. On the contrary, a computer system is inherently discontinuous and the smallest modification of an input can lead to a completely different behavior and break down of the entire system. Computer systems are extremely sensitive to every details. From this standpoint, formal verification is interesting because with a comprehensive proof that is verifiable by a machine, we can make sure that every detail of the computer system has received a fully detailed review of why it should work properly, whereas it is so easy to miss details with an on-paper argumentation.

As a result, formal compiler verification is a trustworthy method to ensure the correctness of the compiler. The safety guarantees that this approach to trustworthy compilation can achieve is particularly important if the compiler is to be used in the development of safety-critical software. Indeed, the safety requirements for the development tools that transform programs are very high and require the most rigorous development methods.

Several ambitious compiler verification efforts are currently under way, such as the Jinja project of Klein and Nipkow [KN03, KN06], the Verisoft project of Leinenbach et al. [Str05, LPP05], and the CompCert project of Leroy et al. [Ler06, L+08].

Translation validation Translation validation, as introduced by Pnueli et al. [PSS98b], provides a systematic way to detect (at compile-time) semantic discrepancies between the input and the output of an optimization. At every compilation run, the input code and the generated

code are fed to a validator (a piece of software distinct from the compiler itself), which tries to establish *a posteriori* that the generated code behaves as prescribed by the input code. If the validator succeeds, compilation proceeds normally. If, however, the validator detects a discrepancy, or is unable to establish the desired semantic equivalence, compilation is aborted; some validators also produce an explanation of the error.

Translation validators are easier to design and implement than formally verified compilers. For that matter, the range of optimizations that current translation validators can validate is much more furnished than that of formal verification. Besides, translation validators generally allow the compiler-implementer to modify the implementation of a compiler pass, for instance to tweak the heuristics that the compiler pass uses. In section 1.4, we present a brief state of the art in translation validators design.

Since the validator can be developed independently from the compiler, and generally uses very different algorithms than those of the compiler, translation validation significantly increases the user's confidence in the compilation process. However, it is possible that a compiler bug still goes unnoticed because of a matching bug in the validator. More pragmatically, translation validators, just like type checkers and bytecode verifiers, are difficult to test: while examples of correct code that should pass abound, building a comprehensive suite of incorrect code that should be rejected is delicate [SB99]. The guarantees obtained by translation validation are therefore weaker than those obtained by formal compiler verification. This is a weakness that we set out to remedy in the present work.

1.2 Formal verification of translation validators

A crucial observation that drives the work presented in this dissertation is that translation validation can provide formal correctness guarantees as strong as those obtained by compiler verification, provided the validator itself is formally verified. In other words, it suffices to model the validator and prove that it implies the desired semantic equivalence result between the source code and the compiled code. The compiler or compiler pass itself does not need to be proved correct and can use algorithms, heuristics and implementation techniques that do not easily lend themselves to program proof. We claim that for many optimization passes, the approach outlined above — translation validation *a posteriori* combined with formal verification of the validator — can be significantly less involved than formal verification of the compilation pass, yet provide the same level of assurance.

To make this claim more precise, we model a compiler or compiler pass as a function $L_1 \to L_2 + \texttt{Error}$, where the `Error` result denotes a compile-time failure, L_1 is the source language and L_2 is the target language for this pass.

Let $\leq$ be a relation between a program $c_1 \in L_1$ and a program $c_2 \in L_2$ that defines the desired semantic preservation property for the compiler pass. We say that a compiler $C : L_1 \to$

$L_2 + \texttt{Error}$ is *formally verified* if we have proved that

$$\forall c_1 \in L_1, c_2 \in L_2, \quad C(c_1) = c_2 \;\Rightarrow\; c_1 \leq c_2 \tag{1}$$

In the translation validation approach, the compiler pass is complemented by a *validator*: a function $L_1 \times L_2 \to \texttt{boolean}$. A validator V is formally verified if we have proved that

$$\forall c_1 \in L_1, c_2 \in L_2, \quad V(c_1, c_2) = \texttt{true} \;\Rightarrow\; c_1 \leq c_2 \tag{2}$$

Let C be a compiler and V a validator. The following function C_V defines a formally verified compiler from L_1 to L_2:

$$\begin{array}{rcl} C_V(c_1) & = & c_2 \;\text{ if } C(c_1) = c_2 \text{ and } V(c_1, c_2) = \texttt{true} \\ C_V(c_1) & = & \texttt{Error} \;\text{ if } C(c_1) = c_2 \text{ and } V(c_1, c_2) = \texttt{false} \\ C_V(c_1) & = & \texttt{Error} \;\text{ if } C(c_1) = \texttt{Error} \end{array}$$

The line of work presented in this dissertation follows from the trivial theorem below.

Theorem 1.1. *If the validator V is formally verified in the sense of (2), then the compiler C_V is formally verified in the sense of (1).*

In other words, the verification effort for the derived compiler C_V reduces to the verification of the validator V. The original compiler C itself does not need to be verified and can be treated as a black box. As compilers are naturally defined as a composition of compilation passes, verified compilers can also be decomposed in this manner. Thus, for every compiler pass of a verified compiler, the compiler writer has the choice between proving the correctness of his compiler pass or using formally verified translation validation.

Consequently, it is possible to use an untrusted implementation within a formally verified compiler. This is useful when it is impossible or too difficult to formalize the transformation within the proof assistant. This may be due to limitations imposed by the logic of the proof assistant; to intellectual property; because part of the transformation requires human intervention; because the transformation is correct only with high probability (for instance if it uses random interpretation [Gul05]) or simply if there exists already a good implementation and its formalization is not desirable.

More importantly, even when it is possible to use the verified transformation approach, formal verification of translation validators can be an interesting alternative. In order to be realistic and interesting it must be possible to design and implement a verified validator that satisfies four requirements. The first two requirements are potential properties of our approach that make it advantageous over the formal verification of the transformation. The last two requirements are show stoppers: if we do not meet them, the result will be useless.

1. **The validator's proof must be simple**. The proof of correctness of the validator should be simpler than that of the transformation itself. Otherwise, having to design a validator is just an extra burden. Fortunately, it is sometimes easier to check the results of an algorithm than to prove the algorithm, even when the checker has to be proved itself. Consider for example the quicksort algorithm. Even though proving its correctness is simple, it is even simpler to write and prove correct a function that checks that a list is sorted and that its elements correspond to that of the original unsorted list. Intuitively, we may be able to take advantage of this with translation validation. However, most previous works have focused on the design of general-purpose validation frameworks whose correctness proof may be involved. If there is one particular optimization of family of optimizations that we want to add to the compiler, a better solution may be to design a validator crafted for this optimization and whose proof may be simpler than that of the optimization itself. We will need to find the adequate design for the optimizations under consideration.

2. **The validator must be robust**. A common criticism against formal verification is that every time the software is modified, the proof must be re-done. In the case of optimizations, this can be troublesome since we may need to use different heuristics or analyses depending on the target architecture or specific need (is it more important to produce a smaller or a faster program ?). Because the algorithms used for the translation validator can be very different from the transformation itself, they may be insensitive to small changes and give us the possibility to experiment with variants. If we take again the example of quicksort, the function that checks if a list is a correct sort of another and its proof of correctness should not depend on the particular sorting algorithm used: the property "*being sorted*" is intrinsic and independent of means of sorting. We have to design our validators such that changing the heuristics or the analyses of the transformation will not need any change in the proof.

3. **The validator must be reasonably complete**. Once we have a set of optimizations in mind that should be within the reach of our validator, it should never be possible that the validator report a semantics discrepancy when there is not. As an extreme example, consider a trivial validator that always return false and abort the compilation. From a semantic preservation point of view, this is correct ! Indeed, all the object codes that are produced have the same semantics as their inputs since nothing is in fact produced... Obviously, this is not what we want. More pragmatically, if the validator rejects too many fine transformations, it makes it useless. In this dissertation, we will not provide any formal proof of completeness but we will try to understand the conditions under which the validator works and make those conditions acceptable.

4. **The validator must incur a reasonable performance cost**. At every compiler run, the validator is called to perform its verification. Therefore, translation validation increases

compilation time. We consider this acceptable as long as the validation time is reasonable, that is, small compared with the transformation time itself. Consequently, we must pay attention not only to the design of the validator, but also to its implementation. As much as possible, we will implement our validators with efficient data structures, explain how they could be improved, study their complexity, and perform benchmarks. Note, however, that this constraint can be relaxed. Indeed, it is not necessary to run the validator at early stages of the development, but only at the very last compilation run, before the code is shipped for its final usage. In such a scenario, it could be acceptable that validation is slow.

These goals may be contradictory. In this dissertation, we privilege proof simplicity and completeness over performance and robustness against variations of the transformation.

To summarize, the thesis we defend in this dissertation is that *it is possible to design and implement formally verified translation validators for realistic and aggressive untrusted optimizations such that the validator is easier to prove than the transformation, robust to slight changes in the transformation, reasonably complete, and has reasonable performance costs.*

1.3 Case studies: formally verified validation of 4 optimizations

To evaluate the concept of formal verification of translation validation, we set out to design, build and prove correct translation validators for a set of optimizations that are both standard in industrial-strengh optimizing compilers and difficult to prove. Intuitively, the more non-local a transformation is, the more difficult it is to prove because the invariant of semantics preservation becomes more complex, relating parts of the codes that may span conditional branches or loops. Thus, by increasing order of complexity, we will make four case studies: list scheduling, trace scheduling, lazy code motion, and finally software pipelining. The optimizations are implemented in the OCaml language, the validators implemented and verified using the Coq proof assistant [BC04, Coq08], and the resulting compiler passes plugged into the Compcert C formally verified compiler. In the remainder of this section, we briefly review the principles, challenges, solution and contributions of each experiment.

List scheduling List scheduling is an optimization that reorders instructions within blocks to reduce pipeline stalls. It doesn't modify the structure of the program and instructions can never be moved across a control instruction.

The challenge presented by this optimization is that, within a block, the relation between the input and the output code states is difficult to define. Consider the first instruction of the input block. It can be placed anywhere in the output block, in particular, it is not necessarily the first instruction of the output block. We must execute both blocks completely before the

two program states match again. If we try to describe exactly where every instruction has gone and how the states will eventually be similar, we end up with a complex semantic preservation invariant and validator.

The key ingredient to address the problem is symbolic evaluation. Symbolic evaluation uses the standard mathematical technique of inventing symbols to represent arbitrary program inputs and then executing a sequence of linear code with those symbolic input values. This execution results in a symbolic execution tree that is another representation of the code but with fewer syntactical details. In particular, symbolic evaluation is unaffected by the relative order of two instructions as long as they are semantically equivalent. This gives us a basis to check whether two sequences of linear code are semantically equivalent. In order to deal with potential runtime errors, symbolic evaluation is extended to gather knowledge about which symbolic trees of the input code are (or must be, for the output code) safe to execute. From this primitive, it is relatively easy to build a validator and prove its correctness since it is simple to reason block by block on the execution.

This first experiment, although carried on a rather simple transformation, shows that translation validation not only is a plausible alternative to compiler verification but also has the expected advantages: once armed with symbolic evaluation, the validator and its proof are a few lines of code and proof. Moreover, they are insensitive to the choice of scheduling heuristics.

Trace scheduling Trace scheduling is an optimization that reorders the instructions within traces to reduce pipeline stalls. Traces are any paths in the control-flow graph that do not cross loop entrances. Unlike list scheduling, trace scheduling moves instructions across block boundaries, as long as they are not loop entrances.

The challenge presented by this optimization is that we cannot simply use symbolic evaluation to hide instruction ordering details, as we have done for list scheduling, since instructions are moved across block boundaries. As instructions are moved along traces, they can cross fork or join points in the control-flow graph. To preserve semantics, these instructions must be duplicated on other paths; this is called bookkeeping. This means that, by moving an instruction within a trace, the other traces that intersect are also modified (instructions are added into those traces). It is thus not possible to reason only on one given trace and it is not clear what is the scope of modification on the graph when one trace is scheduled.

The solution we propose is two-fold. On the one hand, we design a new program representation and semantics where control-flow graph of a function is replaced by a graph of trees. Those trees form a partition of the control-flow graph. An instruction that moves around a trace or that was added because of bookkeeping is guaranteed to remain within the boundaries of a tree. The transformation from one representation to another is verified using translation validation. On the other hand, we extend symbolic evaluation to work on trees of instructions, effectively including conditional branching instructions.

Trace scheduling is not supposed to be applied on every program path: it would make the compilation slow and the resulting program too big. Instead, the traces whose scheduling will bring the best performance improvement must be chosen. This may be done for example using feedback from previous run of the application we compile. This experiment shows that the choice of trace is completely orthogonal from the validation algorithm and proof. It is the same with bookkeeping: the position at which instructions are added does not matter for the validator. Moreover, formalizing trace picking and bookkeeping would add much difficulty. Hence, this experiment provides more evidence of the engineering advantages of formally verified translation validation.

Lazy code motion Lazy code motion is an optimization that eliminates redundant computations on the whole function body: computations that occur multiple times on some paths. It also eliminates partially redundant computations (computations that occur multiple times on some paths and only once on others) and moves loop invariants computations out of loops. Although the structure of the control-flow graph is not modified, the transformation is global because a computation can be moved across forks, joins, and loops of the control-flow graph.

Lazy code motion is considered to be a challenging optimization because it uses a set of four dataflow analyses. The dataflow equations can be difficult to get right and their effect happens on the whole function body, including loops; therefore, a proof of correctness seems difficult because an invariant has to maintain properties on a global scale. Also, because instructions are moved, the question of their well-definedness arise. When the transformation moves a division within the graph, we must make sure that we are not transforming a program with well-defined semantics into a program that produces a runtime error.

Yet the solution that uses translation validation is surprisingly simple. It consists in a single global dataflow analysis: an available expressions analysis. By computing available expressions at every program point, we can verify, when a redundant computation has been replaced by a move from a register that supposedly carries the value of the computation, that the register indeed carry the computation that was performed in the initial code. To answer the concern about runtime errors in new instructions, we designed an algorithm, that we call an anticipability checker, that crawls through the initial code to make sure that the new instruction is safe to execute.

This experiment is a paradigmatic example of formally verified translation validation. The validator is simple and efficient: a simple dataflow analysis with an anticipability checker that is a search through the graph. It is also independent of the choice of dataflow analysis used to perform redundancy elimination. Even more than that, thanks to a careful design, it only requires a small amount of work to verify other transformations such as constant propagation with strengh reduction. The optimization can be implemented to be very efficient, using for example bit vector analysis, which would be unpleasant to formalize.

Software pipelining Software pipelining is an optimization that reorders instructions within loops such that an iteration of the initial loop is spread out in several iterations of the transformed loop, resulting in inter-iterations parallelism. It is followed by modulo variable expansion, which unrolls the loop and renames registers. This transformation changes the structure of loops and instructions are moved across loop boundaries.

The challenge presented by this optimization is that the invariant of semantic preservation is extremely complex. The main reason is that the output loop is a parallel version of the input loop. There is no simple relation between the execution of both loops: instructions have been reordered and span several iterations and the registers used are different.

The key idea to solve the problem is, again, to use symbolic evaluation to state a simple symbolic invariant that holds between the loops. Using symbolic evaluation, and hence hiding syntactical details, we can express an invariant that can be checked. That constitutes the key primitive of our validator. Proving the correctness is not obvious and we had to study symbolic evaluation further and in particular its abstract algebraic structure. Armed with this theory, the proof of correctness becomes short and intuitive.

This experiment shows that, using formally verified translation validation, challenging optimizations are within reach of formal verification. It follows that even the most tricky optimizations are within the reach of formally verified compilers.

1.4 A brief history of translation validation

A cornerstone of the case studies that we present in this dissertation is the design and implementation of translation validators for the transformations considered. There has been a wealth of research on this subject and we now review some of the key works. The first presentation of what is now known as translation validation probably dates back to Samet's Ph.D. thesis in 1975 [Sam75]. In his dissertation, he presents a validator for an optimizing compiler from a subset of Lisp 1.6 down to an assembly language for the PDP-10. An interesting comment from his dissertation explain why he considered this problem:

The main motivation for this thesis has been a desire to write an optimizing compiler for LISP. In order to do this properly, it was determined that a proof was necessary of the correctness of the optimization process. Thus we are especially interested in proving that small perturbations in the code leave the effect of the function unchanged. It is our feeling that this is where the big payoff lies in optimizing LISP.

The concept of translation validation was re-discovered and pushed forward by Pnueli et al [PSS98b, PSS98a] – who also coined the term translation validation – to make the compilation of the synchronous language Signal more trustworthy. Their approach works by generating a set of verification conditions that attest that the compiled program is a refinement of the source program. Those verification conditions are discharged by a theorem prover. Translation

validation also appeared under the name of credible compilation in the work of Rinard and Marinov [RM99]. They propose to validate compilation for sequential languages by generating, along with the output code, a proof that the semantics has been preserved. (The compiler certifies that the compilation was correct.)

The design and implementation of translation validators for optimizing compilers of sequential languages starts off with Necula [Nec00]. He builds a translation validation infrastructure powerful enough to validate a significant part of an industrial strengh optimizing compiler: GCC version 2.7. It makes use of symbolic evaluation and verifies equivalence between programs by collecting and solving constraints.

Zuck et al. [ZPL01, BFG$^+$05, ZPFG03, GZB05, LP06, PZ08] introduce a methodology to validate optimizations using a verification condition generator and a theorem prover. The verification conditions state program equivalence for finite slices of the programs executions. A fundamental idea of their work is that the invariant for an intraprocedural optimization is composed of:

- A relation between the control-flow graphs nodes;
- A relation between the program states resources (registers, memory);
- Invariants that state properties about the individual input and output programs.

Given this information, their tool is able to generate a set of verification conditions that can be discharged by a theorem prover and imply that the output program is a correct refinement of the input program. Of course, it can be hard to find those relations and invariants – it is where we need the properties of the underlying transformation –. Most of the general purpose validators that have been designed since then rely on this idea and most special purpose validators can be recast into this framework. Zuck et al. also extended their framework to handle many loop optimizations and interprocedural optimizations. They have implemented several validators, including one for the SGI Pro-64 compiler and one for the Intel ORC compiler.

Rival [Riv04] validates compilation from a C source down to assembly in one shot, by computing an abstract model of the C program and of the assembly code using symbolic transfer functions – symbolic evaluation extended to handle conditionals – and then generating verification conditions by abstract interpretation that are discharged by a theorem prover. This work is interesting in that it shows that it is possible to validate programs expressed in very different languages by translating then into a common abstract language.

Huang et al. [HCS06] have designed and implemented a special purpose validator for register allocation. This validator uses dataflow analysis to validate the transformation and is able to provide helpful information explaining the cause of the semantic discrepancies that it detects. Their validator is crafted especially for that purpose; therefore, it is efficient and is believed to be both correct and complete for register allocation.

Participants	kind	compiler	validated transformation(s)	note	reference
Necula	GP	GCC 2.7	CSE, loop unrolling/inversion, induction variable, register allocation, instruction scheduling	Symbolic evaluation followed by constraints solving	[Nec00]
Barrett, Fang, Goldberg, Hu, Leviathan, Pnueli, Zaks, Zuck	GP	SGI Pro-64, Intel ORC, DiabData	constant folding, copy propagation, dead code elimination, loop inversion, strengh reduction, loop fusion/distribution/unrolling/peeling/alignment, software pipelining	Generation of verification conditions that imply that the transformed program is a refinement of the original program followed by theorem proving	[ZPL01, BFG^{+}05, ZPFG03, GZB05, LP06, PZ08]
Rival	GP	GCC 3.0	compilation from C down to assembly, without optimizations	Computation of a common representation for the C and the assembly program using symbolic-transfer functions followed by theorem proving to unify both programs	[Riv04]
Childers, Huang, Soffa	SP	MachSUIF	register allocation	Dataflow analysis	[HCS06]
Leroy, Tristan	SP	Compcert	list and trace scheduling, lazy code motion, software pipelining	Symbolic evaluation and dataflow analysis	[TL08, TL09]
Kanade, Khedker, Sanyal	GP	GCC 4.1	loop invariant code motion, partial redundancy elimination, lazy code motion, code hoisting, copy and constant propagation	The compiler is instrumented such that every modification of the control-flow graph is verified to be applicable by the PVS model checker	[KSK06]
Pnueli, Zaks	GP	LLVM	Unknown	Computation of the cross-product of the two programs followed by dataflow analysis	[ZP08]
Kundu, Lerner, Tatlock	GP		all optimizations presented above, except register allocation and instruction scheduling	The optimization are defined in a domain specific language and are validated once and for all with the help of a theorem prover	[KTL09]

Table 1.1: Some the translation validation experiments

Kanade et al [KSK06] present a validation framework based on the idea that many optimizations can be seen, in the end, as a sequence of very primitive rewriting over the control-flow graph. (Primitive rewritings include adding a node, moving a node, changing the instruction at a given node, etc.) In order to preserve semantics, those rewriting must satisfy a few properties that are expressed using modal logic (following the idea that model-checking, abstract interpretation and dataflow analysis are strongly related) and discharged by the PVS model checker.

Kundu et al [KTL09] present parameterized translation validation. In their framework, an optimization is defined as a set of rewrite rules that apply on a partially specified program (the part not specified being any context program into which the specified part can be plugged). The rewrite rules are proved to be correct *once and for all* under some conditions that are verified at compile time using a theorem prover.

Zaks and Pnueli [ZP08] introduce a new general-purpose validator. The key idea is that, by computing the so-called cross product of the two programs that have to be validated, the problem of validation reduces to the problem of analyzing a single program. Therefore, the many techniques that exist to analyze a single program can be used for translation validation.

In table 1.1 we summarize a few key information on previous experiments on the translation validation of optimizing compilers. Algorithms for translation validation roughly fall in two classes. General-purpose validators (GP) such as those of Pnueli et al. [PSS98b], Necula [Nec00], Barret et al. [BFG^{+}05], Rival [Riv04] and Kanade [KSK06] rely on generic techniques such as symbolic execution, model-checking and theorem proving, and can therefore be applied to a wide range of program transformations. Since checking semantic equivalence between two code fragments is undecidable in general, these validators can generate false alarms and have high complexity. If we are interested only in a particular optimization or family of related optimizations, special-purpose validators (SP) can be developed, taking advantage of our knowledge of the limited range of code transformations that these optimizations can perform. An example of a special-purpose validator is that of Huang et al. [HCS06] for register allocation.

Among the range of techniques that are used to design validators, symbolic evaluation plays an important role in this dissertation. We will explain in details what symbolic evaluation is in the following chapters. Despite its extreme simplicity, symbolic evaluation is of first importance in the field of program's proofs. It seems to be almost as old as the idea of mechanically proving the correctness of programs [Kin69, Deu73] and is still used in many recent software verification tools. It is closely related to the concept of generating verification conditions, where the result of symbolic evaluation is used to instantiate a post-condition of a particular program.

1.5 Notes about this document

The Compcert C compiler, and more particularly the intermediate languages that we use, are presented in chapter 2. We discuss list scheduling in chapter 3, trace scheduling in chapter 4, lazy code motion in chapter 5 and software pipelining in chapter 6. We conclude in chapter 7.

Although this dissertation is about formally verified software, the proofs in this dissertation are presented informally. This is a deliberate choice that is justified by the fact that a proof serves two purposes [Geu09]:

- Convince that a statement is correct.
- Explain why the statement is correct.

Otherwise noted, the statements presented in this dissertation have been mechanically proved using the Coq proof assistant. Therefore, we focus our attention to the explanations of how to build validators, how to prove their correctness, why they imply semantic preservation and only sketch proofs when they help the intuition.

The experiments on list scheduling and trace scheduling have been published in the proceedings of the 35th Annual Sigplan-Sigact Symposium on Principles of Programming Languages [TL08] and presented in January 2008 in San Francisco, USA. The experiment on lazy code motion has been published in the proceedings of the ACM Sigplan 2009 Conference on Programming Language Design and Implementation [TL09] and has been presented in June 2009 in Dublin, Ireland. The experiment on software pipelining has been accepted for publication in the 37th Annual Sigplan-Sigact Symposium on Principles of Programming Languages [TL10].

Chapter 2

Setting

In this chapter, we give a comprehensive presentation of the syntax and semantics of the intermediate languages upon which we are working in the remainder of the dissertation. After a general presentation of the Compcert backend (section 2.1) we present the elements of syntax and semantics shared by both intermediate languages (section 2.2). Then we present the RTL language (section 2.3) and the Mach language (section 2.4). In the end, we briefly sketch the experimental protocol used through the dissertation and explain how a formally verified piece of software can be linked with an untrusted implementation (section 2.5).

This chapter serves as a reference chapter. Section 2.3 is useful only for chapters 5 and 6. Section 2.4 is useful only for chapters 3 and 4. The reader who is only interested in understanding the validation algorithms and getting a high-level understanding of the proofs can skip this chapter.

2.1 The Compcert verified compiler

The Compcert compiler is the result of an investigation on formal verification of realistic compilers usable for critical embedded software. The backend of Compcert targets the PowerPC architecture, a common chip for embedded software. There is also an ARM backend. There are two working frontends. One frontend takes as its source a very large subset of the C language. This includes all of C features except the types `long long` and `long double`, statements `goto`, `longjump` and `setjumps`, extreme forms of `switch` such as in Duff's device and finally unprototyped and variable-arity functions. The second frontend is for a pure fragment of an ML-like language.

The architecture of the compiler for the C frontend is depicted in figure 2.1. The backend starts at the Cminor intermediate language. In the backend, the compilation process is as follows:

- Instruction selection and generation of an RTL control-flow graph. (From Cminor to RTL.)

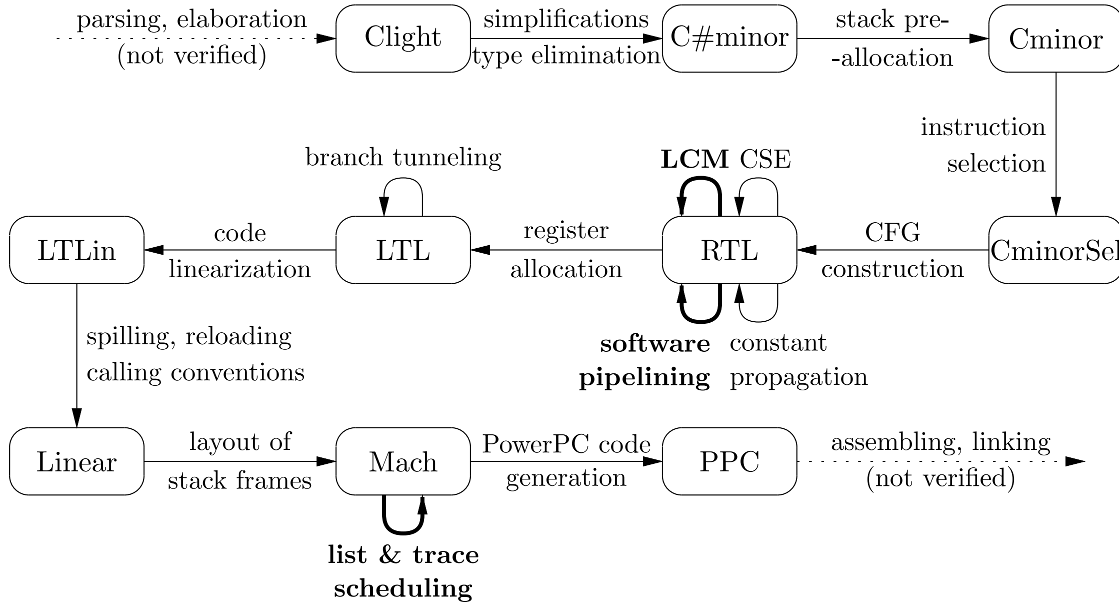


Figure 2.1: Architecture of the Compcert compiler. The thick arrows correspond to the compilation passes presented in this dissertation.

- Two optimizations: Constant propagation with strength reduction and local value numbering. (Over RTL.)
- Register allocation. (From RTL to LTL.)
- One optimization: Branch tunneling. (Over LTL.)
- Linerarization of the code. (From LTL to LTLin.)
- Implementation of spilling, reloading and calling conventions. (From LTLin to Linear.)
- Layout of the stack. (From Linear to Mach.)
- Code generation. (From Mach to PowerPC assembly.)

All of the intermediate languages are formalized using operational semantics. The languages that we use in this dissertation are the Mach language (for list and trace scheduling) and the RTL language (for lazy code motion and software pipelining). During the course of the work presented in this dissertation, the semantics have changed. Hence the Mach semantics is a mixed operational semantics (mostly small step but the function calls are big steps) that models terminating executions. The RTL language semantics is completely small step and models both terminating and diverging executions.

All the intermediate languages of the backend share the memory and events model, the formalization of the global structure of programs and of the operations. We first present those

generalities that are necessary all along the dissertation. Then we present in more details the two intermediate languages that we are going to work on.

2.2 Elements of syntax and semantics

In this section, we present elements of syntax and semantics that are shared amongst the intermediate languages that we use in this dissertation. Those definitions are those of the Compcert compiler: they are not a novelty of this work and appear courtesy of Xavier Leroy. We do not give a fully detailed presentation of the semantics because some details are irrelevant considering the program transformations that we perform. For instance, we only sketch the memory model: our optimizations do not interfere with memory operations, thus, we do not need any commutation properties over memory operations. We refer the reader who needs to pass a judgment on the realism of the Compcert semantics to the project's website [L+08].

2.2.1 Programs

The syntax of RTL and Mach programs share the following common shape.

Programs:

$$
\begin{array}{lll}
P & ::= \{ \ \texttt{vars} = id_1 = data_1^*; \dots id_n = data_n^*; & \text{global variables} \\
 & \quad\ \texttt{functs} = id_1 = Fd_1; \dots id_n = Fd_n; & \text{functions} \\
 & \quad\ \texttt{main} = id \ \} & \text{entry point}
\end{array}
$$

Function definitions:

$$Fd \ ::= \texttt{internal}(F) \mid \texttt{external}(Fe)$$

Definitions of internal functions:

$$F \ ::= \{ \ \texttt{sig} = sig; \texttt{body} = \dots; \dots \ \} \qquad \text{(language-dependent)}$$

Declarations of external functions:

$$Fe \ ::= \{ \ \texttt{name} = id; \texttt{sig} = sig \ \}$$

Initialization data for global variables:

$$
\begin{array}{ll}
data ::= & \texttt{reserve}(n) \mid \texttt{int8}(n) \mid \texttt{int16}(n) \\
 & \mid \texttt{int32}(n) \mid \texttt{float32}(f) \mid \texttt{float64}(f)
\end{array}
$$

Function signatures:

$$sig \ ::= \{ \ \texttt{args} = \vec{\tau}; \ \texttt{res} = (\tau \mid \texttt{void})\}$$

Types:

$$
\begin{array}{lll}
\tau & ::= \texttt{int} & \text{integers and pointers} \\
 & \quad \mid \texttt{float} & \text{floating-point numbers}
\end{array}
$$

A program is composed of global variables with their initialization data along with function definitions. One of the function is designated as the starting function, like in C where the starting function is always `main`.

There are two kinds of functions, internal and external. Internal functions are defined within the program; they are composed of a signature, a function body and other language dependent information. The signature of functions is composed of the types of the arguments of the function along with the type of the return value or `void` when the function does not return a value. The representation of the function body depends on the intermediate language: it can be a list of instructions, a control-flow graph or an abstract syntax tree. External functions are defined by a name and a signature. They model calls to external libraries and system calls. The types of function arguments are either `int` for 32-bit integers and pointers or `float` for 64-bit IEEE 754 floating points.

Given a transformation `transf` over functions, the Compcert compiler provides a function `transp` that generalizes the transformation to programs by applying `transf` on every function of the input program. `transf` may fail on one of the function, resulting in a failure of the overall program transformation.

2.2.2 Values and memory states

The Compcert semantics uses four kinds of values: integers, floats, pointers and `undef` that denotes any value. The memory model [LB08] that Compcert uses models memory as a set of disjoints blocks (by construction). Thus, a pointer is defined by a block b along with an offset δ within the block.

Values:	$v ::= \texttt{int}(n)$	32-bit machine integer
	$\mid \texttt{float}(f)$	64-bit floating-point number
	$\mid \texttt{ptr}(b, \delta)$	pointer
	$\mid \texttt{undef}$	
Memory blocks:	$b \in \mathbb{Z}$	block identifiers
Block offsets:	$\delta ::= n$	byte offset within a block

Each block has lower and upper bounds and associates values to byte offsets within these bounds. The basic operations over memory states are:

- $\texttt{alloc}(M, l, h) = (b, M')$: allocate a fresh block with bounds $[l, h)$, of size $(h - l)$ bytes; return its identifier b and the updated memory state M'.
- $\texttt{store}(M, \kappa, b, \delta, v) = \lfloor M' \rfloor$: store value v in the memory quantity κ of block b at offset δ; return update memory state M'.

- load$(M, \kappa, b, \delta) = \lfloor v \rfloor$: read the value v contained in the memory quantity κ of block b at offset δ.
- free$(M, b) = M'$: free (invalidate) the block b and return the updated memory M'.

Memory operations use the information in κ to define how the chunk of memory being accessed is to be interpreted. For instance, `int8signed` indicates that the bits in memory should be interpreted as an 8-bit signed integer.

Memory quantities: κ ::= `int8signed` | `int8unsigned` | `int16signed`
| `int16unsigned` | `int32` | `float32` | `float64`

The `load` and `store` operations are only partially defined. To be well-defined, a memory-access must be performed on a valid block to an offset that is within the bounds of the block. In the remainder of this dissertation, we use the notation $\lfloor v \rfloor$ to denote a successful computation that returns v and $\emptyset$ to denote a runtime error.

2.2.3 Global environments

In Compcert, data reside in memory blocks whose identifiers are greater than 0 and function codes reside in memory blocks whose identifiers are less than 0. The operational semantics of RTL and Mach are parameterized by a global environment G that does not change during execution. A global environment G maps function blocks $b < 0$ to function definitions and global identifiers to blocks b. The basic operations over global environments are:

- funct$(G, b) = \lfloor Fd \rfloor$: return the function definition Fd corresponding to the block $b < 0$, if any.
- symbol$(G, id) = \lfloor b \rfloor$: return the block b corresponding to the global variable or function name id, if any.
- globalenv$(P) = G$: construct the global environment G associated with the program P.
- initmem$(P) = M$: construct the initial memory state M for executing the program P.

The allocation of blocks for functions and global variables is deterministic so that convenient commutation properties hold between operations on global environments and per-function transformations of programs. Consequently, if a program is transformed function per function, as is the case when using `transp`, the effort of proving semantics preservation for the program essentially reduces to the effort of proving that the semantics is preserved for any function. Since the optimizations we study in this dissertation are intraprocedural, we therefore focus on how to enforce semantic preservation at the level of function codes, the generalization to semantic preservation at the level of programs being obvious.

2.2.4 Traces

The observable behavior of a Compcert program is a sequence of events. An event result from a external function call and is composed of the name of the function called, the arguments of the function and the returned value, if any.

Events:	$\nu ::= id(\vec{v_\nu} \mapsto v_\nu)$	
Event values:	$v_\nu ::= \texttt{int}(n) \mid \texttt{float}(f)$	
Traces:	$t ::= \varepsilon \mid \nu.t$	finite traces (inductive)
	$T ::= \varepsilon \mid \nu.T$	finite or infinite traces (coinductive)
Behaviors:	$B ::= \texttt{converges}(t, n)$	termination with trace t and exit code n
	$\mid \texttt{diverges}(T)$	divergence with trace T

Traces may be finite, written t, or infinite, written T. The former correspond to terminating executions and the latter to diverging executions. Therefore, the observable behavior of a semantically well-defined program is either a finite trace of events plus an exit code or a an infinite trace.

Traces are equipped with concatenation and it is possible for traces to be empty (when no calls to external functions are made). Traces along with the concatenation operator and the neutral element form a monoid.

The following inference rule expresses how a call to an external function *Fe* returns a value v and produces an observable event t, written down as $Fe(\vec{v}) \overset{t}{\looparrowright} v$.

$$\frac{\begin{array}{c}\vec{v} \text{ and } v \text{ are integers or floats}\\ \vec{v} \text{ and } v \text{ agree in number and types with } Fe.\texttt{sig}\\ t = Fe.\texttt{name}(\vec{v} \mapsto v)\end{array}}{Fe(\vec{v}) \overset{t}{\looparrowright} v}$$

2.2.5 Operators, conditions and addressing modes

The machine-specific operators *op* include all common nullary, unary and binary operators of PowerPC, plus immediate forms of many integer operators, as well as a number of combined operators such as not-or, not-and, and rotate-and-mask. ($\texttt{rolm}_{n,m}$ is a left rotation by n bits followed by a logical "and" with m.) There are also the different addressing modes and conditions.

Constants:

cst	$::= n \mid f$	integer or float literal
	$\mid \texttt{addrsymbol}(id)$	address of a global symbol

	$\mid$ `addrstack`(δ)	address within stack data

Comparisons:

c	$::=$ `eq` $\mid$ `ne` $\mid$ `gt` $\mid$ `ge` $\mid$ `lt` $\mid$ `le`	

Unary operators:

op_1	$::=$ `negint` $\mid$ `notbool`	integer arithmetic
	$\mid$ `negf` $\mid$ `absf`	float arithmetic
	$\mid$ `cast8u` $\mid$ `cast8s` $\mid$ `cast16u` $\mid$ `cast16s`	zero and sign extensions
	$\mid$ `singleoffloat`	float truncation
	$\mid$ `intoffloat` $\mid$ `intuoffloat`	float-to-int conversions
	$\mid$ `floatofint` $\mid$ `floatofintu`	int-to-float conversions

Binary operators:

op_2	$::=$ `add` $\mid$ `sub` $\mid$ `mul` $\mid$ `div`	integer arithmetic
	$\mid$ `and` $\mid$ `or` $\mid$ `xor` $\mid$ `shl` $\mid$ `shr` $\mid$ `shru`	integer bit operation
	$\mid$ `addf` $\mid$ `subf` $\mid$ `mulf` $\mid$ `divf`	float arithmetic
	$\mid$ `cmp`(c) $\mid$ `cmpu`(c) $\mid$ `cmpf`(c)	comparisons

PPC operators:

op	$::= cst \mid op_1 \mid op_2$	Classical operators
	$\mid \mathtt{addi}_n \mid \mathtt{rolm}_{n,m} \mid \ldots$	PPC combined operators

Addressing modes (machine-specific):

$mode$	$::=$ `indexed`(n)	indexed, immediate displacement n
	$\mid$ `indexed2`	indexed, register displacement
	$\mid$ `global`(id, δ)	address is $id + \delta$
	$\mid$ `based`(id, δ)	indexed, displacement is $id + \delta$
	$\mid$ `stack`(δ)	address is stack pointer $+ \delta$

Conditions (machine-specific):

$cond$	$::=$ `comp`(c) $\mid$ `compimm`(c, n)	signed integer / pointer comparison
	$\mid$ `compu`(c) $\mid$ `compuimm`(c, n)	unsigned integer comparison
	$\mid$ `compf`(c)	float comparison

The semantics of operators, modes and conditions are given by functions `eval_op`, `eval_mode` and `eval_cond`. We omit their precise definition as it has little impact on the optimizations studied in this dissertation.

2.3 The RTL intermediate language

The RTL intermediate language comes after instruction selection and before register allocation. It is a classical intermediate program representation – in the sense that many compilers use it –

over which many optimizations can be performed.

2.3.1 Syntax

In RTL, function code is represented as a control-flow graph of instructions. RTL instructions correspond roughly to machine instructions with the notable exception that they operate over so called *temporary* registers (sometimes called pseudo-registers). Each function has its own infinite set of temporaries. In the following, r ranges over temporaries and l over labels of CFG nodes.

RTL instructions:	$i ::= \texttt{nop}(l)$	no operation (go to l)
	$\mid \texttt{op}(op, \vec{r}, r, l)$	arithmetic operation
	$\mid \texttt{load}(\kappa, mode, \vec{r}, r, l)$	memory load
	$\mid \texttt{store}(\kappa, mode, \vec{r}, r, l)$	memory store
	$\mid \texttt{call}(sig, (r \mid id), \vec{r}, r, l)$	function call
	$\mid \texttt{tailcall}(sig, (r \mid id), \vec{r})$	function tail call
	$\mid \texttt{cond}(cond, \vec{r}, l_{true}, l_{false})$	conditional branch
	$\mid \texttt{return} \mid \texttt{return}(r)$	function return
RTL control-flow graph:	$g ::= l \mapsto i$	finite map
RTL functions:	$F ::= \{ \texttt{sig} = sig;$	
	$\texttt{params} = \vec{r};$	parameters
	$\texttt{stacksize} = n;$	size of stack data block
	$\texttt{entrypoint} = l;$	label of first instruction
	$\texttt{code} = g\}$	control-flow graph

The arguments of the function are all passed in a list of temporaries and the returned value, if any, is passed in a temporary as well. The instructions operate over temporaries and carry a label that point to the successor of the instruction.

2.3.2 Semantics

The semantics of RTL programs is given as a small-step operational semantics. The program states have the following form:

Program states:	$S ::= \mathcal{S}(\Sigma, g, \sigma, l, R, M)$	regular state
	$\mid \mathcal{C}(\Sigma, Fd, \vec{v}, M)$	call state
	$\mid \mathcal{R}(\Sigma, v, M)$	return state
Call stacks:	$\Sigma ::= (\mathcal{F}(r, F, \sigma, l, R))^*$	list of frames

$$\frac{g(l) = \lfloor \texttt{nop}(l') \rfloor}{G \vdash \mathcal{S}(\Sigma, g, \sigma, l, R, M) \xrightarrow{\varepsilon} \mathcal{S}(\Sigma, g, \sigma, l', R, M)}$$

$$\frac{g(l) = \lfloor \texttt{op}(op, \vec{r}, r, l') \rfloor \quad \texttt{eval_op}(G, \sigma, op, R(\vec{r})) = \lfloor v \rfloor}{G \vdash \mathcal{S}(\Sigma, g, \sigma, l, R, M) \xrightarrow{\varepsilon} \mathcal{S}(\Sigma, g, \sigma, l', R\{r \leftarrow v\}, M)}$$

$$\frac{\begin{array}{c} g(l) = \lfloor \texttt{load}(\kappa, \mathit{mode}, \vec{r}, r, l') \rfloor \\ \texttt{eval_mode}(G, \sigma, \mathit{mode}, R(\vec{r})) = \lfloor \texttt{ptr}(b, \delta) \rfloor \quad \texttt{load}(M, \kappa, b, \delta) = \lfloor v \rfloor \end{array}}{G \vdash \mathcal{S}(\Sigma, g, \sigma, l, R, M) \xrightarrow{\varepsilon} \mathcal{S}(\Sigma, g, \sigma, l', R\{r \leftarrow v\}, M)}$$

$$\frac{\begin{array}{c} g(l) = \lfloor \texttt{store}(\kappa, \mathit{mode}, \vec{r}, r, l') \rfloor \\ \texttt{eval_mode}(G, \sigma, \mathit{mode}, R(\vec{r})) = \lfloor \texttt{ptr}(b, \delta) \rfloor \quad \texttt{store}(M, \kappa, b, \delta, R(r)) = \lfloor M' \rfloor \end{array}}{G \vdash \mathcal{S}(\Sigma, g, \sigma, l, R, M) \xrightarrow{\varepsilon} \mathcal{S}(\Sigma, g, \sigma, l', R, M')}$$

$$\frac{g(l) = \lfloor \texttt{call}(\mathit{sig}, r_f, \vec{r}, r, l') \rfloor \quad R(r_f) = \texttt{ptr}(b, 0) \quad \texttt{funct}(G, b) = \lfloor \mathit{Fd} \rfloor \quad \mathit{Fd}.\texttt{sig} = \mathit{sig}}{G \vdash \mathcal{S}(\Sigma, g, \sigma, l, R, M) \xrightarrow{\varepsilon} \mathcal{C}(\mathcal{F}(r, g, \sigma, l', R).\Sigma, \mathit{Fd}, R(\vec{r}), M)}$$

$$\frac{g(l) = \lfloor \texttt{tailcall}(\mathit{sig}, r_f, \vec{r}) \rfloor \quad R(r_f) = \texttt{ptr}(b, 0) \quad \texttt{funct}(G, b) = \lfloor \mathit{Fd} \rfloor \quad \mathit{Fd}.\texttt{sig} = \mathit{sig}}{G \vdash \mathcal{S}(\Sigma, g, \sigma, l, R, M) \xrightarrow{\varepsilon} \mathcal{C}(\Sigma, \mathit{Fd}, R(\vec{r}), \texttt{free}(M, \sigma))}$$

$$\frac{g(l) = \lfloor \texttt{cond}(\mathit{cond}, \vec{r}, l_{\mathit{true}}, l_{\mathit{false}}) \rfloor \quad \texttt{eval_cond}(\mathit{cond}, R(\vec{r})) = \lfloor \texttt{true} \rfloor}{G \vdash \mathcal{S}(\Sigma, g, \sigma, l, R, M) \xrightarrow{\varepsilon} \mathcal{S}(\Sigma, g, \sigma, l_{\mathit{true}}, R, M)}$$

$$\frac{g(l) = \lfloor \texttt{cond}(\mathit{cond}, \vec{r}, l_{\mathit{true}}, l_{\mathit{false}}) \rfloor \quad \texttt{eval_cond}(\mathit{cond}, R(\vec{r})) = \lfloor \texttt{false} \rfloor}{G \vdash \mathcal{S}(\Sigma, g, \sigma, l, R, M) \xrightarrow{\varepsilon} \mathcal{S}(\Sigma, g, \sigma, l_{\mathit{false}}, R, M)}$$

$$\frac{g(l) = \lfloor \texttt{return} \rfloor}{G \vdash \mathcal{S}(\Sigma, g, \sigma, l, R, M) \xrightarrow{\varepsilon} \mathcal{R}(\Sigma, \texttt{undef}, \texttt{free}(M, \sigma))}$$

$$\frac{g(l) = \lfloor \texttt{return}(r) \rfloor}{G \vdash \mathcal{S}(\Sigma, g, \sigma, l, R, M) \xrightarrow{\varepsilon} \mathcal{R}(\Sigma, R(r), \texttt{free}(M, \sigma))}$$

$$\frac{\texttt{alloc}(M, 0, F.\texttt{stacksize}) = (\sigma, M')}{G \vdash \mathcal{C}(\Sigma, \texttt{internal}(F), \vec{v}, M) \xrightarrow{\varepsilon} \mathcal{S}(\Sigma, F.\texttt{code}, \sigma, F.\texttt{entrypoint}, [F.\texttt{params} \mapsto \vec{v}], M')}$$

$$\frac{\mathit{Fe}(\vec{v}) \overset{t}{\hookrightarrow} v}{G \vdash \mathcal{C}(\Sigma, \texttt{external}(\mathit{Fe}), \vec{v}, M) \xrightarrow{t} \mathcal{R}(\Sigma, v, M)}$$

$$G \vdash \mathcal{R}(\mathcal{F}(r, g, \sigma, l, R).\Sigma, v, M) \xrightarrow{\varepsilon} \mathcal{S}(\Sigma, g, \sigma, l, R[r \leftarrow v], M)$$

$$\frac{\texttt{symbol}(\texttt{globalenv}(P), P.\texttt{main}) = \lfloor b \rfloor \quad \texttt{funct}(\texttt{globalenv}(P), b) = \lfloor \mathit{Fd} \rfloor}{\texttt{initial}(P, \mathcal{C}(\varepsilon, \mathit{Fd}, \varepsilon, \texttt{initmem}(P)))}$$

$$\texttt{final}(\mathcal{R}(\varepsilon, \texttt{int}(n), M), n)$$

Figure 2.2: Semantics of RTL

Register states: $R ::= r \mapsto v$

- Regular states $\mathcal{S}$ correspond to an execution point within an internal function. In regular states, g is the CFG of the function currently executing, l a program point (CFG node label) within this function, σ its stack data block, and R assigns values to the pseudo-registers of F.
- Call states $\mathcal{C}$ materialize parameter passing from the caller to the callee. They carry the function definition Fd being invoked, and a list of argument values.
- Return states $\mathcal{R}$ correspond to returning from a function to its caller. They carry the return value.

All three kinds of states contain the current memory state and the call stack Σ. The call stack is a list of frames (or activation record) $\mathcal{F}$ that represent pending function calls. A frame contains the state of a function necessary to resume its computation: the code F, the stack pointer σ, the program point l, and the register R to which the callee will place the return value.

Initial states are call states with an empty call stack. A call state where the called function is external transitions directly to a return state after generating the appropriate event in the trace. A call state where the called function is internal transitions to a regular state corresponding to the function entry point, possibly after binding the argument values to the parameter variables. Non-call, non-return instructions go from regular states to regular states. A non-tail call instruction resolves the called function, pushes a return frame on the call stack and transitions to the corresponding call state. A tail call is similar but does not push a return frame. A return instruction transitions to a return state. A return state with a non-empty call stack pops the top return frame and moves to the corresponding regular state. A return state with an empty call stack is a final state.

The transition relation between states is written $G \vdash S \xrightarrow{t} S'$. It denotes one execution step from state S to state S' in global environment G that produce the observable event t. Note that the only transition that produce an event that is not ε is the call to an external function.

In addition to the type of states S and the transition relation $G \vdash S \xrightarrow{t} S'$, we have two predicates that express what is a program execution starting state and what is a final state:

- $\texttt{initial}(P, S)$: the state S is an initial state for the program P. S is an invocation of the main function of P in the initial memory state $\texttt{initmem}(P)$.
- $\texttt{final}(S, n)$: the state S is a final state with exit code n. The program is returning from the initial invocation of its main function, with return value $\texttt{int}(n)$.

Executions are modeled classically as sequences of transitions from an initial state to a final state. We write $G \vdash S \xrightarrow{t}{}^{+} S'$ to denote one or several transitions (transitive closure), $G \vdash S \xrightarrow{t}{}^{*}$

S' to denote zero, one or several transitions (reflexive transitive closure), and $G \vdash S \xrightarrow{T} \infty$ to denote an infinite sequence of transitions starting with S. The traces t (finite) and T (finite or infinite) are formed by concatenating the traces of elementary transitions. Formally:

$$G \vdash S \xrightarrow{\varepsilon}{}^{*} S \qquad \frac{G \vdash S \xrightarrow{t_1} S' \quad G \vdash S' \xrightarrow{t_2}{}^{*} S''}{G \vdash S \xrightarrow{t_1.t_2}{}^{*} S''}$$

$$\frac{G \vdash S \xrightarrow{t_1} S' \quad G \vdash S' \xrightarrow{t_2}{}^{*} S''}{G \vdash S \xrightarrow{t_1.t_2}{}^{+} S''} \qquad \frac{G \vdash S \xrightarrow{t} S' \quad G \vdash S' \xrightarrow{T} \infty}{G \vdash S \xrightarrow{t.T} \infty}$$

The inference rule defining a diverging execution $G \vdash S \xrightarrow{T} \infty$ is to be interpreted coinductively, as a greatest fixpoint.

Finally, the observable behavior of a program P is defined as follows. Starting from an initial state, if a finite sequence of reductions with trace t leads to a final state with exit code n, the program has observable behavior $\texttt{converges}(t, n)$. If an infinite sequence of reductions with trace T is possible, the observable behavior of the program is $\texttt{diverges}(T)$.

$$\frac{\texttt{initial}(P, S) \quad \texttt{globalenv}(P) \vdash S \xrightarrow{t}{}^{*} S' \quad \texttt{final}(S', n)}{P \Downarrow \texttt{converges}(t, n)}$$

$$\frac{\texttt{initial}(P, S) \quad \texttt{globalenv}(P) \vdash S \xrightarrow{T} \infty}{P \Downarrow \texttt{diverges}(T)}$$

2.4 The Mach intermediate language

The Mach language is the last intermediate language before the generation of PPC assembly code. In this language, the activation record has been laid out and can be accessed through three additional instructions: $\texttt{setstack}(r, \tau, \delta)$, $\texttt{getstack}(\tau, \delta, r)$, $\texttt{getparent}(\tau, \delta, r)$. τ is the type of the data moved and δ its word offset in the corresponding activation record. `setstack` and `getstack` write and read within the activation record of the current function. `getparent` reads within the activation record of the caller.

The basic operations over a frame are:

- $\texttt{get_slot}(F, \tau, \delta) = \lfloor v \rfloor$: read value v in the activation record F at offset δ.
- $\texttt{set_slot}(F, \tau, \delta, v) = \lfloor F' \rfloor$: store value v in activation record F at offset δ; returns the new activation record F'.

2.4.1 Syntax

The Mach language represents functions as a list of abstract instructions, corresponding roughly to machine instructions. Thus, control is implemented using labels and the semantics use the partial function $\texttt{find_label}(l, c)$ which returns the sub-list of the code c starting at l, if any. To the contrary of RTL instructions, Mach instructions operate over machine registers.

Functions carry two byte offsets, `retaddr` and `link`, indicating where in the activation record the function prologue should save the return address into its caller and the back link to the activation record of its caller, respectively.

Machine registers: $r_m ::= \texttt{R3} \mid \texttt{R4} \mid \ldots$ PowerPC integer registers
$\mid \texttt{F1} \mid \texttt{F2} \mid \ldots$ PowerPC float registers

Mach instructions:

$i ::= \texttt{setstack}(r, \tau, \delta)$	register to stack move
$\mid \texttt{getstack}(\tau, \delta, r)$	stack to register move
$\mid \texttt{getparent}(\tau, \delta, r)$	caller's stack to register move
$\mid \texttt{op}(op, \vec{r}, r)$	arithmetic operation
$\mid \texttt{load}(\kappa, mode, \vec{r}, r)$	memory load
$\mid \texttt{store}(\kappa, mode, \vec{r}, r)$	memory store
$\mid \texttt{call}(sig, (r \mid id))$	function call
$\mid \texttt{cond}(cond, \vec{r}, l_{true})$	conditional branch
$\mid \texttt{goto}(l)$	unconditional branch
$\mid \texttt{label}(l)$	definition of the label l
$\mid \texttt{return}$	function return

Linear code sequences:

$c ::= i_1 \ldots i_n$	list of instructions

Mach functions:

$F ::= \{ \texttt{sig} = sig;$	
$\texttt{stack_high} = n;$	upper bound of stack data block
$\texttt{stack_low} = n;$	lower bound of stack data block
$\texttt{retaddr} = \delta;$	offset of saved return address
$\texttt{link} = \delta;$	offset of back link
$\texttt{code} = c\}$	instructions

2.4.2 Semantics

The Mach semantics is a mix of big-step, for function calls, and small-step, for the other instructions. In the small-step rules, a state is composed of the activation record Σ of the function, a

$$\frac{\texttt{set_slot}(F, \tau, \delta, R(r)) = \lfloor F' \rfloor}{G, F_n, \sigma, \psi \vdash \texttt{setstack}(r, \tau, \delta); c, R, F, M \xrightarrow{\varepsilon} c, R, F', M}$$

$$\frac{\texttt{get_slot}(F, \tau, \delta) = \lfloor v \rfloor}{G, F_n, \sigma, \psi \vdash \texttt{getstack}(\tau, \delta, r); c, R, F, M \xrightarrow{\varepsilon} c, R\{r \leftarrow v\}, F, M}$$

$$\frac{\texttt{get_slot}(\psi, \tau, \delta) = \lfloor v \rfloor}{G, F_n, \sigma, \psi \vdash \texttt{getparent}(\tau, \delta, r); c, R, F, M \xrightarrow{\varepsilon} c, R\{r \leftarrow v\}, F, M}$$

$$\frac{R(r) = \texttt{ptr}(b, 0) \quad \texttt{funct}(G, b) = \lfloor Fd \rfloor \quad Fd.\texttt{sig} = sig \quad (G, Fd, F) \vdash (R, M) \overset{t}{\Rightarrow} (R', M')}{G, F_n, \sigma, \psi \vdash \texttt{call}(sig, r); c, R, F, M \xrightarrow{t} c, R', F, M'}$$

$$\frac{\texttt{eval_op}(G, \sigma, op, R(\vec{r})) = \lfloor v \rfloor}{G, F_n, \sigma, \psi \vdash \texttt{op}(op, \vec{r}, r); c, R, F, M \xrightarrow{\varepsilon} c, R\{r \leftarrow v\}, F, M}$$

$$\frac{\texttt{eval_mode}(G, \sigma, mode, R(\vec{r})) = \lfloor \texttt{ptr}(b, \delta) \rfloor \quad \texttt{load}(M, \kappa, b, \delta) = \lfloor v \rfloor}{G, F_n, \sigma, \psi \vdash \texttt{load}(\kappa, mode, \vec{r}, r); c, R, F, M \xrightarrow{\varepsilon} c, R\{r \leftarrow v\}, F, M}$$

$$\frac{\texttt{eval_mode}(G, \sigma, mode, R(\vec{r})) = \lfloor \texttt{ptr}(b, \delta) \rfloor \quad \texttt{store}(M, \kappa, b, \delta, R(r)) = \lfloor M' \rfloor}{G, F_n, \sigma, \psi \vdash \texttt{store}(\kappa, mode, \vec{r}, r); c, R, F, M \xrightarrow{\varepsilon} c, R, F, M'}$$

$$G, F_n, \sigma, \psi \vdash \texttt{label}(l); c, R, F, M \xrightarrow{\varepsilon} c, R, F, M$$

$$\frac{\texttt{find_label}(l, f.\texttt{code}) = \lfloor c' \rfloor}{G, F_n, \sigma, \psi \vdash \texttt{goto}(l); c, R, F, M \xrightarrow{\varepsilon} c', R, F, M}$$

$$\frac{\texttt{eval_cond}(cond, R(\vec{r})) = \lfloor \texttt{true} \rfloor \quad \texttt{find_label}(l_{true}, f.\texttt{code}) = \lfloor c' \rfloor}{G, F_n, \sigma, \psi \vdash \texttt{cond}(cond, \vec{r}, l_{true}); c, R, F, M \xrightarrow{\varepsilon} c', R\{r \leftarrow v\}, F, M}$$

$$\frac{\texttt{eval_cond}(cond, R(\vec{r})) = \lfloor \texttt{false} \rfloor}{G, F_n, \sigma, \psi \vdash \texttt{cond}(cond, \vec{r}, l_{true}); c, R, F, M \xrightarrow{\varepsilon} c, R\{r \leftarrow v\}, F, M}$$

Figure 2.3: Semantics of Mach instructions

sequence of code to execute c, a register file R and the memory M. The environment, sometimes written Σ, is composed of the global environment G, the function being executed F_n, the stack pointer σ and the activation record of the caller ψ. In the big-step rules, the state is composed of the register file and the memory and the environment contains the global environment G, the function being executed F and the activation record of the caller ψ.

The relation transitions for all instructions except calls are written $G, F_n, \sigma, \psi \vdash S \xrightarrow{t} S'$ and denotes one execution step from state S to state S' in global environment G, executing function F with stack pointer σ and caller activation record ψ. The trace t denotes the observable events generated by this execution step. The rules for most instructions are similar to the rules of RTL semantics. The notable difference with RTL is the execution of the `call` instruction: the execution of the call is modeled by a one step reduction. It is defined by using another relation transition $G, F, \psi \vdash S \overset{t}{\Rightarrow} S'$ which models, in a natural style the execution of the called function. The execution of the function body itself is modeled as a sequence of small-step transitions (the reflexive and transitive closure of the single-step rule) that begins with the code of the function and ends when a `return` instruction is reached. Thus, the semantics is defined in a mutually inductive way.

$$G, F_n, \sigma, \psi \vdash S \xrightarrow{\varepsilon}{}^{*} S \qquad \frac{G, F_n, \sigma, \psi \vdash S \xrightarrow{t} S'}{G, F_n, \sigma, \psi \vdash S \xrightarrow{t}{}^{*} S'}$$

$$\frac{G, F_n, \sigma, \psi \vdash S \xrightarrow{t_1}{}^{*} S' \qquad G, F_n, \sigma, \psi \vdash S' \xrightarrow{t_2}{}^{*} S''}{G, F_n, \sigma, \psi \vdash S \xrightarrow{t_1.t_2}{}^{*} S''}$$

$$\frac{\begin{array}{c}\forall link, \forall ra, \mathtt{alloc}(M, 0, Fd.\mathtt{stack_high}) = \lfloor (M', \rho) \rfloor \qquad \sigma = \mathtt{ptr}(\rho, Fd.\mathtt{stack_low}) \\ \mathtt{set_slot}(\mathtt{init_frame}, \mathtt{int}, 0, link) = \lfloor F_1 \rfloor \qquad \mathtt{set_slot}(F_1, \mathtt{int}, 12, ra) = \lfloor F_2 \rfloor \\ G, Fd, \sigma, \psi \vdash (F_2, Fd.\mathtt{code}, R, M') \xrightarrow{t}{}^{*} (F_3, \mathtt{return} :: c, R', M'')\end{array}}{G, \mathtt{internal}(Fd), \psi \vdash (R, M) \overset{t}{\Rightarrow} (R', (\mathtt{free}(\rho, M'')))}$$

$$\frac{Fe(\vec{v}) \overset{t}{\hookrightarrow} v}{G, \mathtt{external}(Fe), \psi \vdash (R, M) \overset{t}{\Rightarrow} (R', M')}$$

The observable behavior of a program P is defined as the execution of the program main function that terminates with a trace t and result n. To the contrary of RTL, only converging executions are modeled. The Mach language described here comes from an older version of Compcert which only modeled converging executions.

$$\frac{\texttt{initial}(P,S) \quad \texttt{funct}(G,\texttt{main}) = \lfloor F_n \rfloor \quad \texttt{globalenv}(P), F_n, \texttt{empty_frame} \vdash S \overset{t}{\Rightarrow} S'}{P \Downarrow (t, S'.R(R_3))}$$

2.5 Experimental protocol

All the optimizations that we study in this dissertation were implemented in OCaml. All the validators – except the one for software pipelining – were automatically extracted to Ocaml code from the Coq development and linked with the hand-written implementations of the optimizations. Figure 2.4 shows how the verified compiler is built in Compcert and with the verified validator approach.

Technically, a black box optimization can be declared in the Coq development using the `Parameter` keyword. For instance, the following command declares the existence of a transformation `f` on the RTL intermediate language.

```
Parameter f : RTL -> RTL.
```

Within the Coq development, all that is known about `f` is its type. When the formally verified code is extracted from the Coq development, this function must be ”linked” with the OCaml implementation. (Therefore, the OCaml implementation must have the same type as `f`.) This is done by configuring the Coq extractor with the following command:

```
Extract Constant f => "f_ocaml".
```

where `f_ocaml` is the OCaml implementation.

The tests of the resulting compilers that we present in this dissertation were conducted on a Pentium 4 3.4 GHz Linux machine with 2 GB of RAM. Each pass was repeated as many times as needed for the measured time to be above 1 second; the times reported are averages.

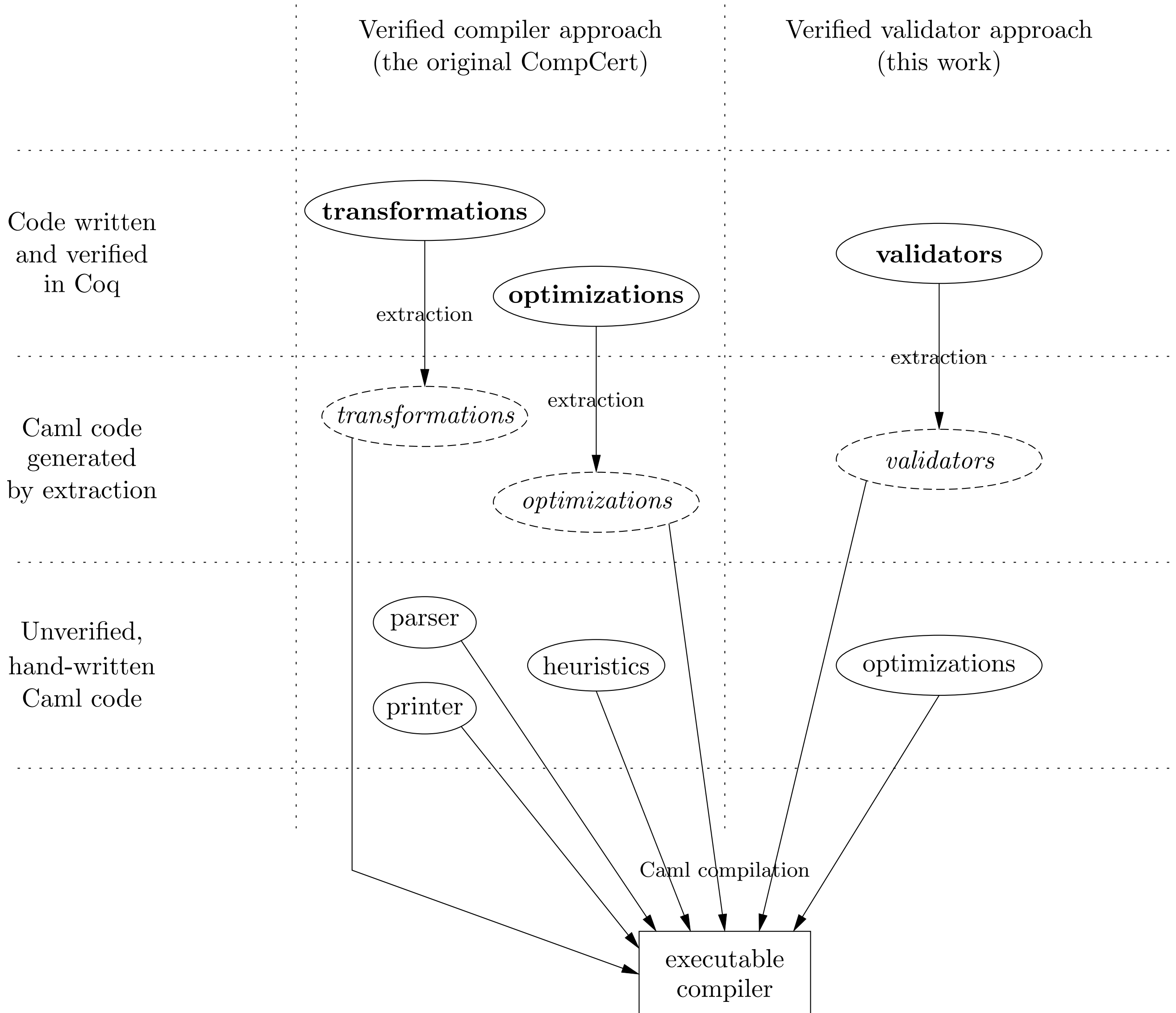


Figure 2.4: Compiler construction. On the left, the usual compiler construction process of Compcert. On the right, the compiler construction with verified validators and untrusted implementation of optimizations.

Chapter 3

List scheduling

In this chapter we study the formal verification of a validator for an optimization called list scheduling. It is a good candidate for a first formal verification because it preserves the block structure of the program: we can therefore focus on the problem of verifying that two linear sequences of code have the same semantics. To this end, we introduce and formalize the concept of *symbolic evaluation*, a fundamental tool for translation validation. Although the validator is simple, the proof is not completely straightforward because it is not clear what the invariant of semantics preservation is during the execution of two corresponding blocks. We solve this problem by giving a new semantics, proved to be equivalent, to the language that encodes the fact that transformation is limited to blocks.

3.1 List scheduling

List scheduling is the simplest instruction scheduling optimization. It takes place at the Mach level. Like all optimizations of its kind, it reorders instructions in the program to increase instruction-level parallelism, by taking advantage of pipelining and multiple functional units. In order to preserve program semantics, reorderings of instructions must respect the following rules, where ρ is a *resource* of the processor (e.g. a register, or the memory):

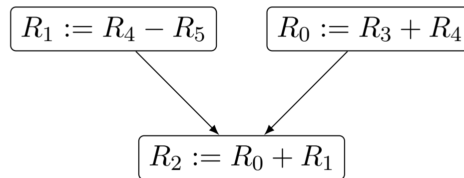
- Write-After-Read: a read from ρ must not be moved after a write to ρ;
- Read-After-Write: a write to ρ must not be moved after a read from ρ;
- Write-After-Write: a write to ρ must not be moved after another write to ρ.

As an example, the two following lists of instructions have the same semantics but the instructions are computed in a different order. This is possible because the two first instructions of

each list are independent of each other.

$R_0 := R_3 + R_4$	$R_1 := R_4 - R_5$
$R_1 := R_4 - R_5$	$R_0 := R_3 + R_4$
$R_2 := R_0 + R_1$	$R_2 := R_0 + R_1$

Below is the data dependency graph of the first list of instructions. An arrow between two instructions denotes a dependency on some machine resource (register or memory). The data dependency graph makes apparent the fact that the second list of instructions is another possible scheduling of the first list of instructions.



We do not detail the implementation of list scheduling, which can be found in compiler textbooks [App98, Muc97]. One important feature of this transformation is that it is performed at the level of basic blocks: instructions are reordered within basic blocks, but never moved across branch instructions nor across labels. Therefore, the control-flow graph of the original and scheduled codes are isomorphic, and translation validation for list scheduling can be performed by comparing matching blocks in the original and scheduled codes.

In the remainder of the chapter we will use the term “block” to denote the longest sequence of non-branching instructions between two branching instructions. The branching instructions in Mach are `label`, `goto`, `cond` and `call`. This is a change from the common view where a block includes its terminating branching instruction.

3.2 Formalization of symbolic evaluation

Following Necula [Nec00], we use *symbolic evaluation* as our main tool to show semantic equivalence between code fragments. Symbolic evaluation of a basic block represents the values of variables at the end of the block as symbolic expressions involving the values of the variables at the beginning of the block. For instance, the symbolic evaluation of

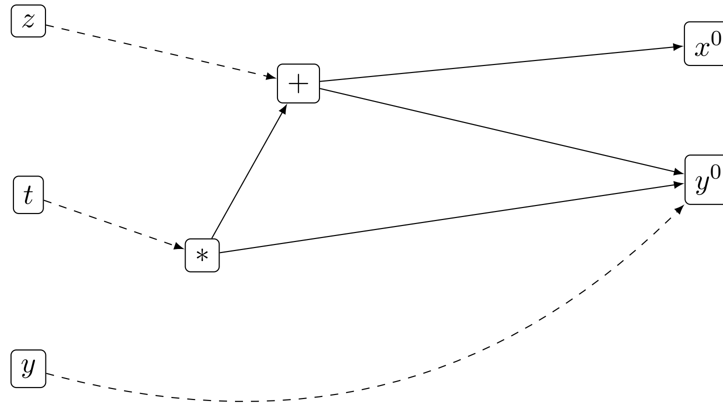
```
z := x + y;
t := z × y
```

is the following mapping of variables to expressions

$$\begin{array}{rcl} \mathtt{z} & \mapsto & \mathtt{x}^0 + \mathtt{y}^0 \\ \mathtt{t} & \mapsto & (\mathtt{x}^0 + \mathtt{y}^0) \times \mathtt{y}^0 \\ v & \mapsto & v^0 \text{ for all other variables } v \end{array}$$

where v^0 symbolically denotes the initial value of variable v at the beginning of the block.

We can also depict the result of the symbolic evaluation of those two instructions as an acyclic graph with sharing of the sub-expressions. We only show the symbolic value associated to z, t and y (their value is pointed to by a dashed arrow).



Symbolic evaluation extends to memory operations if we consider that they operate over an implicit argument and result, `Mem`, representing the current memory state. For instance, the symbolic evaluation of

```
store(x, 12);
y := load(x)
```

is

$$\begin{array}{rcl} \mathtt{Mem} & \mapsto & \mathtt{store}(\mathtt{Mem}^0, \mathtt{x}^0, 12) \\ \mathtt{y} & \mapsto & \mathtt{load}(\mathtt{store}(\mathtt{Mem}^0, \mathtt{x}^0, 12), \mathtt{x}^0) \\ v & \mapsto & v^0 \text{ for all other variables } v \end{array}$$

The crucial observation is that two basic blocks that have the same symbolic evaluation (identical variables are mapped to identical symbolic expressions) are semantically equivalent, in the following sense: if both blocks successfully execute from an initial state Σ, leading to final states Σ_1 and Σ_2 respectively, then $\Sigma_1 = \Sigma_2$.

Necula [Nec00] goes further and compares the symbolic evaluations of the two code fragments modulo equations such as computation of arithmetic operations (e.g. $1 + 2 = 3$), algebraic properties of these operations (e.g. $x + y = y + x$ or $x \times 4 = x \ll 2$), and "good variable" properties for memory accesses (e.g. $\texttt{load}(\texttt{store}(m, p, v), p) = v$). This is necessary to validate transformations such as constant propagation or instruction strength reduction. However, for the instruction scheduling optimizations that we consider here, equations are not necessary and it suffices to compare symbolic expressions by structure.

The semantic equivalence result that we obtain between blocks having identical symbolic evaluations is too weak for our purposes: it does not guarantee that the transformed block executes without run-time errors whenever the original block does. Consider:

```
x := 1          x := x / 0
                x := 1
```

Both blocks have the same symbolic evaluation, namely $\texttt{x} \mapsto 1$ and $v \mapsto v^0$ if $v \neq \texttt{x}$. However, the rightmost block crashes at run-time on a division by 0, and is therefore not a valid optimization of the leftmost block, which does not crash. To address this issue, we enrich symbolic evaluation as follows: in addition to computing a mapping from variables to expressions representing the final state, we also maintain a set of all arithmetic operations and memory accesses performed within the block, represented along with their arguments as expressions. Such expressions, meaning "this computation is well defined", are called *constraints*. In the example above, the set of constraints is empty for the leftmost code, and equal to $\{\texttt{x}^0/0\}$ for the rightmost code. Figure 3.1 summarizes the potential sources of runtime errors in the semantics.

To validate the transformation of a block b_1 into a block b_2, we now do the following: perform symbolic evaluation over b_1, obtaining a mapping m_1 and a set of constraints s_1; do the same for b_2, obtaining m_2, s_2; check that $m_2 = m_1$ and $s_2 \subseteq s_1$. This will guarantee that b_2 executes successfully whenever b_1 does, and moreover the final states will be identical.

3.2.1 Symbolic expressions

The syntax of symbolic expressions that we use is as follows:

Resources:

$\rho \quad ::= r \mid \texttt{Mem} \mid \texttt{Frame}$

Value expressions:

$t \quad ::= r^0$ — initial value of register r

$\mid \texttt{Getstack}(\tau, \delta, t_f)$

$\mid \texttt{Getparent}(\tau, \delta)$

$\mid \texttt{Op}(op, \vec{t})$

$\mid \texttt{Load}(\kappa, mode, \vec{t}, t_m)$

Cause of partial definition of an instruction	Affected instructions
The corresponding processor instruction can fail at runtime	Integer division if dividend is 0; memory load and store from an invalid address
The behavior of the corresponding processor instruction is machine-dependent	Integer shifts by 32 bits or more
Not definable within our memory model	Subtraction or comparisons $<$, $>$, $\leq$, $\geq$ between pointers belonging to different memory blocks
Wrong number of arguments; arguments of the wrong types	Nearly every instruction

Table 3.1: Sources of runtime errors. Many instructions are formalized as partial functions, and can therefore lead to a runtime error in the semantics. The last category of partiality source could be avoided by using a stronger type system.

Memory expressions:

$$
\begin{array}{rll}
t_m & ::= \texttt{Mem}^0 & \text{initial memory store} \\
 & \mid \texttt{Store}(\kappa, mode, \vec{t}, t_m, t) &
\end{array}
$$

Frame expressions:

$$
\begin{array}{rll}
t_f & ::= \texttt{Frame}^0 & \text{initial frame} \\
 & \mid \texttt{Setstack}(t, \tau, \delta, t_f) &
\end{array}
$$

Symbolic code:

$$m \ ::= \rho \mapsto (t \mid t_m \mid t_f)$$

Constraints:

$$s \ ::= \{t, t_m, t_f, \ldots\}$$

The resources we track are the processor registers (tracked individually), the memory state (tracked as a whole), and the *frame* for the current function (the part of its activation record that is treated as separate from the memory by the Mach semantics). The symbolic code m obtained by symbolic evaluation is represented as a map from resources to symbolic expressions t, t_f and t_m of the appropriate kind. Additionally, we also collect a set s of symbolic expressions that have well-defined semantics.

We now give a denotational semantics to symbolic codes, as transformers over concrete states

(R, F, M). We define inductively the following four predicates:

$$\begin{array}{lll} \Sigma \vdash & [\![t]\!](R,F,M) = v & \text{Value expressions} \\ \Sigma \vdash & [\![t_f]\!](R,F,M) = F' & \text{Frame expressions} \\ \Sigma \vdash & [\![t_m]\!](R,F,M) = M' & \text{Memory expressions} \\ \Sigma \vdash & [\![m]\!](R,F,M) = (R',F',M') & \text{Symbolic codes} \end{array}$$

The definition of these predicates is straightforward. We show four selected rules.

$$\frac{\Sigma \vdash [\![\vec{t}]\!](R,F,M) = \vec{v} \qquad \texttt{eval_op}(op, \vec{v}) = v}{\Sigma \vdash [\![\texttt{op}(op, \vec{t})]\!](R,F,M) = v}$$

$$\frac{\begin{array}{c}\Sigma \vdash [\![\vec{t}]\!](R,F,M) = \vec{v} \qquad \Sigma \vdash [\![t_m]\!](R,F,M) = \vec{m} \\ \texttt{eval_mode}(\Sigma.G, \Sigma.\sigma, mode, \vec{v}) = \lfloor \texttt{ptr}(b,\delta) \rfloor \qquad \texttt{load}(m, \kappa, b, \delta) = \lfloor v \rfloor\end{array}}{\Sigma \vdash [\![\texttt{load}(\kappa, mode, \vec{t}, t_m)]\!](R,F,M) = v}$$

$$\frac{\begin{array}{c}\Sigma \vdash [\![t]\!](R,F,M) = v \qquad \Sigma \vdash [\![t_f]\!](R,F,M) = f \\ \texttt{set_slot}(f, \tau, \delta, v) = \lfloor F' \rfloor\end{array}}{\Sigma \vdash [\![\texttt{setstack}(t, \tau, \delta, t_f)]\!](R,F,M) = F'}$$

$$\frac{\begin{array}{c}\forall r, \Sigma \vdash [\![m(r)]\!](R,F,M) = R'(r) \\ \Sigma \vdash [\![m(\texttt{Frame})]\!](R,F,M) = F' \\ \Sigma \vdash [\![m(\texttt{Mem})]\!](R,F,M) = M'\end{array}}{\Sigma \vdash [\![m]\!](R,F,M) = (R',F',M')}$$

For constraints, we say that a symbolic expression t viewed as the constraint "t has well-defined semantics" is satisfied in a concrete state (R, F, M), and we write $\Sigma, (R, F, M) \models t$, if there exists a value v such that $\Sigma \vdash [\![t]\!](R, F, M) = v$, and similarly for symbolic expressions t_f and t_m over frames and memory. For a set of constraints s, we write $\Sigma, (R, F, M) \models s$ if every constraint in s is satisfied in state (R, F, M).

3.2.2 Algorithm for symbolic evaluation

We now give the algorithm that, given a list b of non-branching Mach instructions, computes its symbolic evaluation $\alpha(b) = (m, s)$.

We first define the symbolic evaluation $\alpha(i, (m, s))$ of one instruction i as a transformer from the pair (m, s) of symbolic code and constraints "before" the evaluation of i to the pair (m', s') "after" the evaluation of i. The definition use the function `update`$(\rho, t, (m, s))$: it assigns the symbolic value t to resource ρ in the symbolic code m and adds t to the set of constraints s.

$$
\begin{aligned}
&\texttt{update}(\rho, t, (m, s)) \\
&\quad = \quad (m\{\rho \leftarrow t\},\ s \cup \{t\}) \\
&\alpha(\texttt{setstack}(r, \tau, \delta), (m, s)) \\
&\quad = \quad \texttt{update}(\texttt{Frame}, \texttt{Setstack}(m(r), \tau, \delta, m(\texttt{Frame})), (m, s)) \\
&\alpha(\texttt{getstack}(\tau, \delta, r), (m, s)) \\
&\quad = \quad \texttt{update}(r, \texttt{Getstack}(\tau, \delta, m(\texttt{Frame})), (m, s)) \\
&\alpha(\texttt{getparent}(\tau, \delta, r), (m, s)) \\
&\quad = \quad \texttt{update}(r, \texttt{Getparent}(\tau, \delta), (m, s)) \\
&\alpha(\texttt{op}(op, \vec{r}, r), (m, s)) \\
&\quad = \quad \texttt{update}(r, \texttt{Op}(op, m(\vec{r})), (m, s)) \\
&\alpha(\texttt{load}(\kappa, mode, \vec{r}, r), (m, s)) \\
&\quad = \quad \texttt{update}(r, \texttt{Load}(\kappa, mode, m(\vec{r}), m(\texttt{Mem})), (m, s)) \\
&\alpha(\texttt{store}(\kappa, mode, \vec{r}, r), (m, s)) \\
&\quad = \quad \texttt{update}(\texttt{Mem}, \texttt{Store}(\kappa, mode, m(\vec{r}), m(\texttt{Mem}), m(r)), (m, s))
\end{aligned}
$$

We then define the symbolic evaluation of the block $b = i_1; \ldots; i_n$ by iterating the one-instruction symbolic evaluation function α, starting with the initial symbolic code $\varepsilon = (\rho \mapsto \rho^0)$ and the empty set of constraints.

$$\alpha(b) = \alpha(i_n, \ldots \alpha(i_2, \alpha(i_1, (\varepsilon, \emptyset))) \ldots)$$

Note that all operations performed by the block are recorded in the constraint set s. It is possible to omit operations that cannot fail at run-time (such as "load constant" operators) from s; we elected not to do so for simplicity.

3.2.3 Properties of symbolic evaluation

The symbolic evaluation algorithm has the following two properties that are used later in the proof of correctness for the validator. First, any concrete execution of a block b satisfies its symbolic evaluation $\alpha(b)$, in the following sense.

Lemma 3.1. *Let b be a block and c an instruction list starting with a branching instruction. If $\Sigma \vdash (b; c), R, F, M \xrightarrow{*} c, R', F', M'$ and $\alpha(b) = (m, s)$, then $\Sigma \vdash [\![m]\!](R, F, M) = (R', F', M')$ and $\Sigma, (R, F, M) \models s$.*

Proof. First, we prove that $\Sigma \vdash [\![m]\!](R, F, M) = (R', F', M')$. It is a consequence of the more

general theorem:
If $\Sigma \vdash [\![m_0]\!](R_0, F_0, M_0) = (R, F, M)$ and $\Sigma \vdash (b; c), R, F, M \xrightarrow{*} c, R', F', M'$
then $\Sigma \vdash [\![\alpha(b, m_0)]\!](R_0, F_0, M_0) = (R', F', M')$.
We prove this theorem by induction on b. (Note that, in the induction hypothesis, m and (R_0, F_0, M_0) are universally quantified.) The kernel of the proof is thus:
If $\Sigma \vdash [\![m_0]\!](R_0, F_0, M_0) = (R, F, M)$ and $\Sigma \vdash a, R, F, M \xrightarrow{\varepsilon} c, R', F', M'$
then $\Sigma \vdash [\![\alpha(a, m_0)]\!](R_0, F_0, M_0) = (R', F', M')$
where a is a single instruction and which is proved by a case analysis on a.
Second, we prove that $\Sigma, (R, F, M) \models s$. It can be proved directly by a reversed induction on b. The kernel of the proof is thus:
If $\Sigma \vdash (b; a; c), R, F, M \xrightarrow{*} c, R', F', M'$ and $\Sigma, (R, F, M) \models \texttt{snd}(\alpha(b))$
then $\Sigma, (R, F, M) \models \texttt{snd}(\alpha(b; a))$
where a is an instruction and which is proved by a case analysis on a. □

Second, if an initial state R, F, M satisfies the constraint part of $\alpha(b)$, it is possible to execute b to completion from this initial state.

Lemma 3.2. *Let b be a block and c an instruction list starting with a branching instruction. Let $\alpha(b) = (m, s)$. If $\Sigma, (R, F, M) \models s$, then there exists R', F', M' such that $\Sigma \vdash (b; c), R, F, M \xrightarrow{*} c, R', F', M'$.*

Proof. A key lemma to prove this theorem is:
If $\Sigma, (R, F, M) \models \texttt{snd}(\alpha(b; a))$ then $\Sigma, (R, F, M) \models \texttt{snd}(\alpha(b))$ where a is an instruction.
Then, the proof is by reversed induction on b. From the induction hypothesis, we conclude that there exists a state R_i, F_i, M_i such that for all c, $\Sigma \vdash (b; c), R, F, M \xrightarrow{*} c, R_i, F_i, M_i$ and it remains to prove that the execution of a in state R_i, F_i, M_i is well-defined by a case analysis on a. □

3.3 A validator for list scheduling

We design a validator for list scheduling in two steps. First, we write a validation function that verifies semantics preservation for matching pairs of blocks. Second, based on the block validator, we write a validation function for a whole function.

3.3.1 Validation at the level of blocks

Based on the symbolic evaluation algorithm above, we now define a validator for transformations over blocks. This is a function V_b taking two blocks (lists of non-branching instructions) b_1, b_2 and returning `true` if b_2 is a correct scheduling of b_1.

$V_b(b_1, b_2) =$
    let $(m_1, s_1) = \alpha(b_1)$
    let $(m_2, s_2) = \alpha(b_2)$
    return $m_2 = m_1 \wedge s_2 \subseteq s_1$

3.3.2 Validation at the level of function bodies

Given two lists of instructions c_1 and c_2 corresponding to the body of a function before and after instruction scheduling, the following validator V checks that $V_b(b_1, b_2) =$ `true` for each pair of matching blocks b_1, b_2, and that matching branching instructions are equal. (We require, without significant loss of generality, that the external implementation of list scheduling preserves the order of basic blocks within the function code.)

$V(c_1, c_2) =$
  if c_1 and c_2 are empty:
    return `true`
  if c_1 and c_2 start with a branching instruction:
    decompose c_1 as $i_1 :: c_1'$
    decompose c_2 as $i_2 :: c_2'$
    return $i_1 = i_2 \wedge V(c_1', c_2')$
  if c_1 and c_2 start with a non-branching instruction:
    decompose c_1 as $b_1; c_1'$, where b_1 is a maximal block
    decompose c_2 as $b_2; c_2'$, where b_2 is a maximal block
    return $V_b(b_1, b_2) \wedge V(c_1', c_2')$
  otherwise:
    return `false`

3.4 Proof of correctness

The proof of correctness follows the two-phase structure of the validator. We first prove that the block validator implies semantics preservation at the level of blocks. Then, we prove that the function validation implies semantic preservation at the function level.

3.4.1 Correctness of block validation

The correctness of this validator follows from the properties of symbolic evaluation.

Lemma 3.3. *Let b_1, b_2 be two blocks and c_1, c_2 two instruction sequences starting with branching instructions. If $V_b(b_1, b_2) =$ `true` and $\Sigma \vdash (b_1; c_1), R, F, M \xrightarrow{*} c_1, R', F', M'$, then $\Sigma \vdash (b_2; c_2), R, F, M \xrightarrow{*} c_2, R', F', M'$.*

Proof. Let $(m_1, s_1) = \alpha(b_1)$ and $(m_2, s_2) = \alpha(b_2)$. By hypothesis $V_b(b_1, b_2) = \texttt{true}$, we have $m_2 = m_1$ and $s_2 \subseteq s_1$. By Lemma 3.1, the hypothesis $\Sigma \vdash (b_1; c_1), R, F, M \xrightarrow{*} c_1, R', F', M'$ implies that $\Sigma, (R, F, M) \models s_1$. Since $s_2 \subseteq s_1$, it follows that $\Sigma, (R, F, M) \models s_2$. Therefore, by Lemma 3.2, there exists R'', F'', M'' such that $\Sigma \vdash (b_2; c_2), R, F, M \xrightarrow{*} c_2, R'', F'', M''$. Applying Lemma 3.1 to the evaluations of $(b_1; c_1)$ and $(b_2; c_2)$, we obtain that $\Sigma \vdash [\![m_1]\!](R, F, M) = (R', F', M')$ and $\Sigma \vdash [\![m_2]\!](R, F, M) = (R'', F'', M'')$. Since $m_2 = m_1$ and the denotation of a symbolic code is unique if it exists, it follows that $(R'', F'', M'') = (R', F', M')$. The expected result follows. □

3.4.2 Correctness of function bodies validation

To prove that $V(c_1, c_2) = \texttt{true}$ implies a semantic preservation result between c_1 and c_2, the natural approach is to reason by induction on an execution derivation for c_1. However, such an induction decomposes the execution of c_1 into executions of individual instructions; this is a poor match for the structure of the validation function V, which decomposes c_1 into maximal blocks joined by branching instructions. To bridge this gap, we define an alternate, block-oriented operational semantics for Mach that describes executions as sequences of sub-executions of blocks and of branching instructions. Writing Σ for global contexts and S, S' for quadruples (c, R, F, M), the block-oriented semantics refines the $\Sigma \vdash S \xrightarrow{\varepsilon} S'$, $\Sigma \vdash S \xrightarrow{*} S'$ and $G \vdash F, P, R, M \Rightarrow R', M'$ predicates of the original semantics into the following 5 predicates:

$$
\begin{array}{lll}
\Sigma & \vdash & S \xrightarrow{\varepsilon}_{nb} S' \qquad \text{one non-branching instruction} \\
\Sigma & \vdash & S \xrightarrow{*}_{nb} S' \qquad \text{several non-branching instructions} \\
\Sigma & \vdash & S \xrightarrow{\varepsilon}_{b} S' \qquad \text{one branching instruction} \\
\Sigma & \vdash & S \leadsto S' \qquad \text{block-branch-block sequences} \\
G & \vdash & F, P, R, M \Rightarrow_{blocks} R', M'
\end{array}
$$

The fourth predicate, written $\leadsto$, represents sequences of $\xrightarrow{*}_{nb}$ transitions separated by $\xrightarrow{\varepsilon}_{b}$ transitions:

$$
\frac{\Sigma \vdash S \xrightarrow{*}_{nb} S'}{\Sigma \vdash S \leadsto S'}
$$

$$
\frac{\Sigma \vdash S \xrightarrow{*}_{nb} S_1 \qquad \Sigma \vdash S_1 \xrightarrow{\varepsilon}_{b} S_2 \qquad \Sigma \vdash S_2 \leadsto S'}{\Sigma \vdash S \leadsto S'}
$$

It is easy to show that the $\leadsto$ block-oriented semantics is equivalent to the original $\xrightarrow{*}$ semantics for executions of whole functions.

Lemma 3.4. *$G \vdash F, P, R, M \Rightarrow R', M'$ if and only if $G \vdash F, P, R, M \Rightarrow_{blocks} R', M'$.*

We are now in a position to state and prove the correctness of the validator V. Let p be a program and p' the corresponding program after list scheduling and validation: p' is identical to p except for function bodies, and $V(p(id).\texttt{code}, p'(id).\texttt{code}) = \texttt{true}$ for all function names $id \in p$.

Theorem 3.1. *Let G and G' be the global environments associated with p and p', respectively. If $G \vdash F, P, R, M \Rightarrow R', M'$ and $V(F.\texttt{code}, F'.\texttt{code}) = \texttt{true}$, then $G' \vdash F', P, R, M \Rightarrow R', M'$.*

Proof. We show the analogous result using $\Rightarrow_{blocks}$ instead of $\Rightarrow$ in the premise and conclusion by induction over the evaluation derivation, using Lemma 3.3 to deal with execution of blocks. We conclude by Lemma 3.4. □

3.5 Discussion

We now discuss the implementation and evaluation of the validator we have presented.

3.5.1 Implementation

The validator has been implemented in the Coq proof-assistant version 8.1. It accounts for approximately 1700 lines of Coq code and specifications and 3000 lines of proof. Table 3.2 is a detailed line count showing, for each component of the validator, the size of the specifications (*i.e.* the algorithms and the semantics) and the size of the proofs. The list scheduling transformation has been implemented in OCaml. It took approximately six person-months to build the transformation, the validator and its proof.

	Specifications	Proofs	Total
Symbolic evaluation	736	1079	1815
Block validation	348	1053	1401
Block semantics	190	150	340
Block scheduling validation	264	590	854
Typing	114	149	263
Total	1652	3021	4673

Table 3.2: Size of the development (in non-blank lines of code, without comments)

3.5.2 Experimental evaluation and Complexity analysis

The verified compiler pass that we obtain has been integrated in the Compcert compiler (a version of the Compiler that is older than the one distributed) and tested on the test suite.

All tests were successfully scheduled and validated after scheduling. Manual inspection of the scheduled code reveals that the schedulers performed a fair number of instruction reorderings.

To assess the compile-time overheads introduced by validation, we measured the execution times of the two scheduling transformations and of the corresponding validators. Table 3.3 presents the results.

On all tests except AES, the time spent in validation is comparable to that spent in the non-verified scheduling transformation. The total time (transformation + validation) of instruction scheduling is about 10% of the whole compilation time. The AES test (the optimized reference implementation of the AES encryption algorithm) demonstrates some inefficiencies in our implementation of validation, which takes about 10 times longer than the corresponding transformation.

Test program	List scheduling		
	Transformation	Validation	Ratio V/T
`fib`	0.29 ms	0.47 ms	1.60
`integr`	0.91 ms	0.87 ms	0.96
`qsort`	1.3 ms	1.5 ms	1.15
`fft`	9.1 ms	18 ms	1.98
`sha1`	9.4 ms	6.7 ms	0.71
`aes`	56 ms	550 ms	9.76
`almabench`	25 ms	16 ms	0.65
`stopcopy`	4.1 ms	4.1 ms	1.00
`marksweep`	5.3 ms	6.3 ms	1.18

Table 3.3: Compilation times and verification times

Indeed, our algorithm for symbolic evaluation has a potential source of inefficiency. It is the comparison between the symbolic codes and constraint sets generated by symbolic execution. Viewed as a tree, the symbolic code for a block of length n can contain up to 2^n nodes (consider for instance the block $r_1 = r_0 + r_0; \ldots ; r_n = r_{n-1} + r_{n-1}$). Viewed as a DAG, however, the symbolic code has size linear in the length n of the block, and can be constructed in linear time. However, the comparison function between symbolic codes that we defined in Coq compares symbolic codes as trees, ignoring sharing, and can therefore take $O(2^n)$ time. Using a hash-consed representation for symbolic expressions would lead to much better performance: construction of the symbolic code would take time $O(n \log n)$ (the $\log n$ accounts for the overhead of hash consing), comparison between symbolic codes could be done in time $O(1)$, and inclusion between sets of constraints in time $O(n \log n)$. We haven't been able to implement this solution by lack of an existing Coq library for hash-consed data structures.

3.5.3 A note on diverging executions

The Mach semantics used in this dissertation only models terminating executions. Consequently, we have only proved that our validator enforces semantic preservation for terminating executions, and we do not know if this property holds for diverging executions. However, we believe that the proof that we present in this chapter can be generalized to take into account divergence, without any need to change the validator itself.

Since we impose that blocks do not contain functions calls or loop back edges (or any form of control), a diverging execution is an infinite sequence of execution of finite blocks separated by control instructions. Therefore, semantic preservation between the execution of a program and its optimized version should follow from the fact that there is semantic preservation for each of their block and that their control is syntactically equivalent.

3.5.4 Conclusion

The validator for list scheduling we have presented is, we believe, the first fully mechanized verification of a translation validator. It shows that it is feasible to prove semantics preservation by mean of a verified translation validator. Although this validator was designed with list scheduling in mind, it seems to be applicable to a wider class of code transformations: those that reorder, factor out or duplicate instructions within basic blocks without taking advantage of non-aliasing information. We believe (without any formal proof) that our validator is complete, that is, raises no false alarms for this class of transformations.

It is interesting to note that the validation algorithms proceed very differently from the code transformations that they validate. The validators uses notions such as symbolic execution and block-based decompositions of program executions that have obvious semantic meanings. In contrast, the optimizations rely on notions such as RAW/WAR/WAW dependencies whose semantic meaning is much less obvious. For this reason, we believe (without experience to substantiate this claim) that it would be significantly more difficult to prove directly the correctness of list scheduling. Besides, the validation algorithm is independent of the heuristics used to schedule the instructions.

Chapter 4

Trace scheduling

In this chapter we study the formal verification of a validator for an optimization called trace scheduling. It is a natural follow up of the formal verification of list scheduling because the scope of transformation extends from blocks boundaries to traces (or regions) and modifies the block structure of the program (but not its loop structure). We present a slight improvement of the symbolic evaluation that allows us to reason about conditional branches. We also push further the idea of giving a more appropriate semantics to the language to prove semantics preservation: not only we change the presentation of the semantics but also design a new program representation as graphs of trees. This allows the verification and proof to be very simple but requires to prove the correctness of the change of representation and semantics.

4.1 Trace scheduling

Trace scheduling [Ell86] is a generalization of list scheduling where instructions are allowed to move past a branch or before a join point, as long as this branch or join point does not correspond to a loop back-edge. It takes place at the Mach level. In this work we restrict the instructions that can be moved to non-branching instructions, thus considering a slightly weaker version of trace scheduling than the classical one.

Moving instructions to different basic blocks requires compensating code to be inserted in the control-flow graph, as depicted in figure 4.1. Consider an instruction i that is moved after a conditional instruction targeting a label l in case the condition is `true` (left). Then, in order to preserve the semantics, we must ensure that if the condition is true during execution the instruction i is executed. We insert a "stub", *i.e.* we hijack the control by making the conditional point to a new label l' where the instruction i is executed before going back to the label l.

Dually, consider an instruction i that is moved before a label l targeted by some instruction `goto` (l) (right part of figure 4.1). To ensure semantics preservation, we must hijack the control

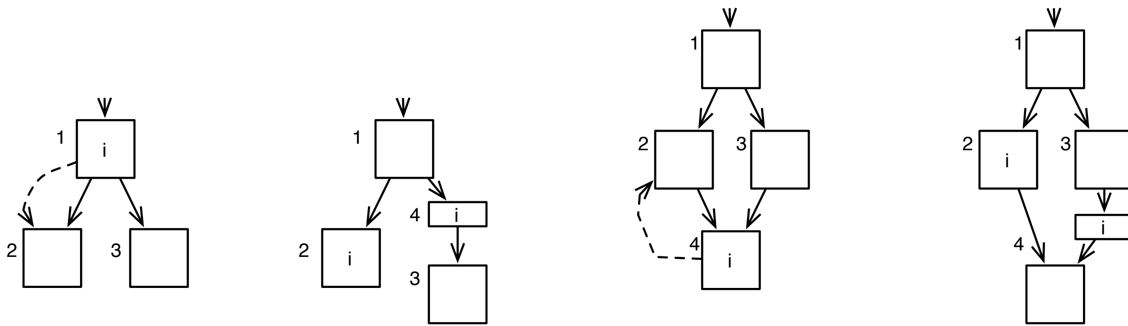


Figure 4.1: The two extra rules of trace scheduling. On the left, an example of move after a condition. On the right, an example of move before a join point. On each example the code is shown before and after hijacking.

of the `goto` into a new stub that contains the instruction i. This way, i is executed even if we enter the trace by following the `goto`.

In list scheduling (chapter 3), the extent of code modifications was limited: an instruction can only move within the basic block that contains it. The unit of modification was therefore the block, i.e. the longest sequences of non-branching instructions between branches. During validation, the branches can then be used as "synchronization points" at which we check that the semantics are preserved. What are the synchronization points for trace scheduling? The only instructions that limit code movement are the return instructions and the target of back-edges, i.e. in our setting, a subset of the labels. We also fix the convention that `call` instructions cannot be crossed. For convenience, a simple pass over the list of instructions add labels in front of return and call instructions. We call those labels "special". Those instructions are our synchronization points, we recall which instructions are considered as a synchronization point or not in the following table:

synchronization points	other instructions
	`op`, `store`, `load`
`call`	`setstack`, `getstack`, `getparent`
`label`(l)	`label`(l)
if l is a back-edge or a special label	if l is not a back-edge nor a special label
`return`	`goto`, `cond`

In conclusion, the unit of modification for trace scheduling is the longest sequence of instructions between these synchronization points.

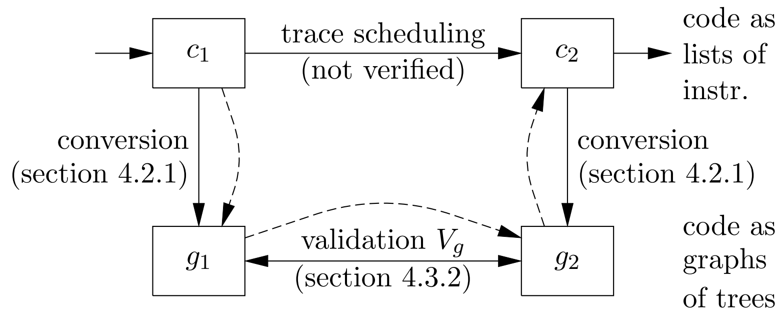


Figure 4.2: Overview of trace scheduling and its validation. Solid arrows represent code transformations and validations. Dashed arrows represent proofs of semantic preservation.

4.2 A tree-based representation of control and its semantics

As in the case of list scheduling, we would like to build the validator in two steps: first, build a function that validates pairs of traces that are expected to match; then, extend it to a validator for whole function bodies. The problem is that a trace can contain branching instructions. The block validator of chapter 3 does not handle this. Moreover, we must ensure that control flows the same way in the two programs, which was obvious for the block validator since states were equivalent before branching instructions, but is no longer true for trace scheduling because of the insertion of compensating code along some control edges.

A solution to these problems is to consider another representation of the program where traces can be manipulated as easily as blocks could in the list-of-instructions representation. This representation is a graph of trees, each tree being a compact representation of all the traces eligible for scheduling that begin at a synchronization point in the control-flow graph. The important property of these trees is that if an instruction has been moved then it must be within the boundaries of a tree.

The validator for trace scheduling is built using this program representation. To complete the validator we must transform our program given as a list of instructions into a semantically equivalent control-flow graph of trees. This leads to the architecture depicted in figure 4.2 that we will detail in the remainder of this section. Note that the transformation from lists of instructions to graphs of trees needs to be proved semantics-preserving in both directions: if the list c is transformed to graph g, it must be the case that g executes from state S to state S' if *and only if* c executes from S to S'.

Figure 4.3 illustrates our tree-based representation of the code of a function. In this section, we formally define its syntax and semantics.

Syntax The code of a function is represented as a mapping from labels to trees. Each label corresponds to a synchronization point in the control-flow graph of the function. A node of a tree is labeled either by a non-branching instruction, with one child representing its unique successor; or by a conditional instruction, with two childs for its two successors. The leaves of

label l_0
op$_1$
cond $\ldots, l_1$
label l_2
op$_2$
cond $\ldots, l_2$
label l_1
op$_3$
label l_4
ret

$$
\begin{aligned}
l_0 &\mapsto \texttt{seq}(\texttt{op}_1, \texttt{cond}(\ldots, \texttt{seq}(\texttt{op}_3, \texttt{out}(l_4)), \\
&\qquad\qquad \texttt{out}(l_2))) \\
l_2 &\mapsto \texttt{seq}(\texttt{op}_2, \texttt{cond}(\ldots, \texttt{seq}(\texttt{op}_3, \texttt{out}(l_4)), \\
&\qquad\qquad \texttt{out}(l_2))) \\
l_4 &\mapsto \texttt{return}
\end{aligned}
$$

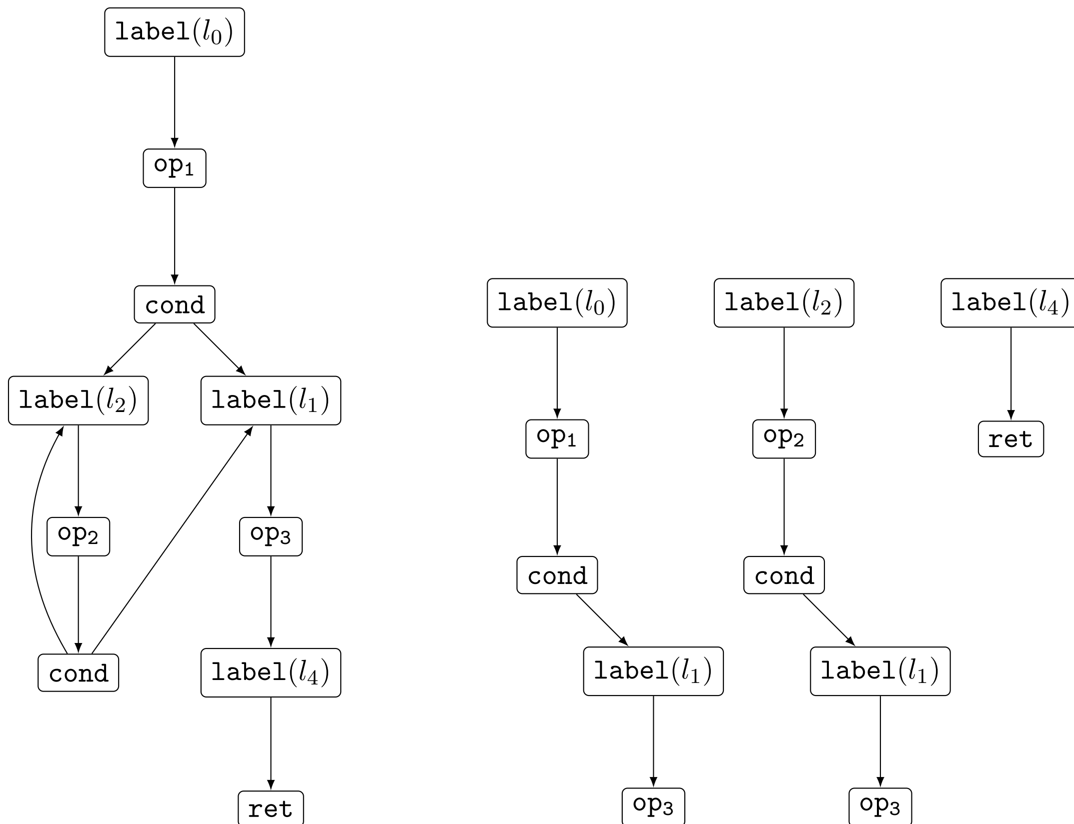


Figure 4.3: A code represented as a list of instructions (upper left), as a graph of instruction trees (upper right) and as a control-flow graph (lower left) along with its trees (lower right).

instruction trees are $\texttt{out}(l)$ nodes, carrying the label l of the tree to which control is transferred. Finally, special one-element trees are introduced to represent call and return instructions.

Instruction trees:

$$\begin{aligned} T \ ::= \ & \texttt{seq}(i, T) && (i \text{ a non-branching instruction}) \\ \mid \ & \texttt{cond}(cond, \vec{r}, T_1, T_2) \\ \mid \ & \texttt{out}(l) \end{aligned}$$

Call trees:

$$T_c ::= \texttt{call}((r \mid id), l)$$

Return trees:

$$T_r ::= \texttt{return}$$

Control-flow graphs:

$$g \ ::= l \mapsto (T \mid T_c \mid T_r)$$

Functions:

$$F \ ::= \texttt{fun}\ id \quad \{\ \texttt{stack}\ n_1; \texttt{frame}\ n_2; \texttt{entry}\ l; \texttt{code}\ g;\ \}$$

Semantics The operational semantics of the tree-based representation is a combination of small-step and big-step styles. We describe executions of instruction trees using a big-step semantics $\Sigma \vdash T, R, F, M \Rightarrow l, R', F', M'$, meaning that the tree T, starting in state (R, F, M), terminates on a branch to label l in state (R', F', M'). Since the execution of a tree cannot loop infinitely, this choice of semantics is adequate, and moreover is a good match for the validation algorithm operating at the level of trees that we develop next. We show three representative rules.

$$\Sigma \vdash \texttt{out}(l), R, F, M \Rightarrow l, R, F, M$$

$$\frac{v = \texttt{eval_op}(op, R(\vec{r})) \qquad \Sigma \vdash T, R\{r_d \leftarrow v\}, F, M \Rightarrow l, R', F', M'}{\Sigma \vdash \texttt{seq}(\texttt{op}(op, \vec{r}, r), T), R, F, M \Rightarrow l, R', F', M'}$$

$$\frac{\texttt{true} = \texttt{eval_condition}(cond, R(\vec{r})) \qquad \Sigma \vdash T_1, R, F, M \Rightarrow l', R', F', M'}{\Sigma \vdash \texttt{cond}(cond, \vec{r}, T_1, T_2), R, F, M \Rightarrow l', R', F', M'}$$

The predicate $\Sigma \vdash l, R, F, M \overset{*}{\rightarrow}_t l', R', F', M'$, defined in small-step style, expresses the

chained evaluation of zero, one or several trees, starting at label l and ending at label l'.

$$\Sigma \vdash l, R, F, M \stackrel{*}{\rightarrow}_t l, R, F, M$$

$$\frac{\Sigma \vdash F.\texttt{graph}(l), R, F, M \Rightarrow l', R', F', M' \qquad \Sigma \vdash l', R', F', M' \stackrel{*}{\rightarrow}_t l'', R'', F'', M''}{\Sigma \vdash l, R, F, M \stackrel{*}{\rightarrow}_t l'', R'', F'', M''}$$

Finally, the predicate for evaluation of function calls, $G \vdash F, P, R, M \Rightarrow v, R', M'$, is redefined in terms of trees in the obvious manner.

$$\frac{\begin{array}{c}\texttt{alloc}(M, 0, F.\texttt{stack}) = (sp, M_1) \quad \texttt{init_frame}(F.\texttt{frame}) = F_1 \quad F.\texttt{graph} = g \\ G, f, sp, P \vdash g(F.\texttt{entry}), R, M_1 \stackrel{*}{\rightarrow}_t l, R', M_2 \quad g(l) = \texttt{return} \quad M' = \texttt{free}(M_2, sp)\end{array}}{G \vdash F, P, R, M \Rightarrow_{traces} R', M'}$$

4.2.1 Conversion to tree-based representation

The conversion algorithm is conceptually simple, but not entirely trivial. In particular, it involves the computation of back edges in order to determine the synchronization points. Instead of writing the conversion algorithm in Coq and proving directly its correctness, we chose to use the translation validation approach one more time. In other terms, the conversion from lists of instructions to graphs of trees is written in unverified Caml, and complemented with a validator, written and proved in Coq, which takes a Mach function F (with its code represented as a list of instructions) and a graph of trees g and checks that $F.\texttt{code}$ and g are semantically equivalent. This check is written $F.\texttt{code} \sim g$.

To check that an instruction sequence C and a graph g are equivalent ($C \sim g$), we enumerate the synchronization points $l \in \text{Dom}(g)$ and check that the list c of instructions starting at point l in the instruction sequence C corresponds to the tree $g(l)$. We write this check as a predicate $C, \mathcal{B} \vdash c \sim T$, where $\mathcal{B} = \text{Dom}(g)$ is the set of synchronization points. The intuition behind this check is that every possible execution path in c should correspond to a path in T that executes the same instructions. In particular, if c starts with a non-branching instruction, we have

$$\frac{i \text{ non-branching} \qquad C, \mathcal{B} \vdash c \sim T}{C, \mathcal{B} \vdash i :: c \sim \texttt{seq}(i, T)}$$

Unconditional and conditional branches appearing in c need special handling. If the target l of the branch is a synchronization point ($l \in \mathcal{B}$), this branch terminates the current trace and

enters a new trace; it must therefore corresponds to an $\texttt{out}(l)$ tree.

$$\frac{l \in \mathcal{B}}{C, \mathcal{B} \vdash \texttt{label}(l) :: c \sim \texttt{out}(l)}$$

$$\frac{l \in \mathcal{B}}{C, \mathcal{B} \vdash \texttt{goto}(l) :: c \sim \texttt{out}(l)}$$

$$\frac{l_{true} \in \mathcal{B} \qquad C, \mathcal{B} \vdash c \sim T}{C, \mathcal{B} \vdash \texttt{cond}(cond, \vec{r}, l_{true}) :: c \sim \texttt{cond}(cond, \vec{r}, \texttt{out}(l_{true}), T)}$$

However, if l is not a synchronization point ($l \notin \mathcal{B}$), the branch or label in c is not materialized in the tree T and is just skipped.

$$\frac{l \notin \mathcal{B} \qquad C, \mathcal{B} \vdash c \sim T}{C, \mathcal{B} \vdash \texttt{label}(l) :: c \sim T}$$

$$\frac{l \notin \mathcal{B} \quad c' = \texttt{find_label}(l, C) \quad C, \mathcal{B} \vdash c' \sim T}{C, \mathcal{B} \vdash \texttt{goto}(l) :: c \sim T}$$

$$\frac{\begin{array}{c} l_{true} \notin \mathcal{B} \qquad c' = \texttt{find_label}(l_{true}, C) \\ C, \mathcal{B} \vdash c \sim T \quad C, \mathcal{B} \vdash c' \sim T' \end{array}}{C, \mathcal{B} \vdash \texttt{cond}(cond, \vec{r}, l_{true}) :: c \sim \texttt{cond}(cond, \vec{r}, T', T)}$$

An interesting fact is that the predicate $C, \mathcal{B} \vdash c \sim T$ indirectly checks that $\mathcal{B}$ contains at least all the targets of back-edges in the code C. For if this were not the case, the code C would contain a loop that does not go through any synchronization point, and we would have to apply one of the three "skip" rules above an infinite number of times; therefore, the inductive predicate $C, \mathcal{B} \vdash c \sim T$ could not hold. As discussed in section 4.5.1, the implementation of the $\sim$ check uses a counter of instructions traversed to abort validation instead of diverging in the case where $\mathcal{B}$ incorrectly fails to account for all back-edges.

The `skip_control` function skips the labels that are not in B and the `goto`s in a list of instructions

$\texttt{skip_control}(B, f, c, k) =$
  if $k = 0$:
    return $\emptyset$
  otherwise, if $c = \texttt{label}(l) :: c'$:

if $l \in B$ return $\lfloor \texttt{label}(l) :: c' \rfloor$
else return $\texttt{skip_control}(B, f, c', (k-1))$
if $c = \texttt{goto}(l) :: c'$:
if $\texttt{find_label}(l, f) = \lfloor c'' \rfloor$
if $l \in B$ return $\lfloor \texttt{goto} l :: c' \rfloor$
else return $\texttt{skip_control}(B, f, c'', (k-1))$
otherwise return $\emptyset$
if $c = i :: c'$:
return $\lfloor i :: c' \rfloor$
in all other cases:
return $\emptyset$

The `test_out` function verifies that a given tree is an `out` node that points to some given label.

$\texttt{test_out}(sub, l) =$
if $sub = \texttt{out}(\texttt{l}')$:
return $l = l'$
otherwise:
return `false`

The `validTreeBase` function implements the $\sim$ check.

`validTreeBase` (B,f,cur,t) =
let $cur' = \texttt{skip_control}(B, (fn_code f), cur, (length(fn_code f))$ in
if $cur' = \texttt{getstack}(i, t, m) :: l$ and $t = getstack(i', t', m', sub)$:
return $i = i' \wedge t = t' \wedge m = m' \wedge \texttt{validTreeBase}(B, f, l, sub)$
if $cur' = \texttt{setstack}(m, i, t) :: l$ and $t = \texttt{setstack}(m', i', t', sub)$:
return $i = i' \wedge t = t' \wedge m = m' \wedge \texttt{validTreeBase}(B, f, l, sub)$
if $cur' = \texttt{getparam}(i, t, m) :: l$ and $t = \texttt{getparam}(i', t', m', sub)$:
return $i = i' \wedge t = t' \wedge m = m' \wedge \texttt{validTreeBase}(B, f, l, sub)$
if $cur' = \texttt{op}(op, lr, m) :: l$ and $t = \texttt{op}(op', lr', m', sub)$:
return $op = op' \wedge lr = lr' \wedge m = m' \wedge \texttt{validTreeBase}(B, f, l, sub)$
if $cur' = \texttt{load}(chk, addr, lr, m) :: l$ and $t = \texttt{load}(chk', addr', lr', m', sub)$:
return $addr = addr' \wedge chk = chk' \wedge lr = lr' \wedge m = m' \wedge \texttt{validTreeBase}(B, f, l, sub)$
if $cur' = \texttt{store}(chk, addr, lr, m) :: l$ and $t = \texttt{store}(chk', addr', lr', m', sub)$:
return $addr = addr' \wedge chk = chk' \wedge lr = lr' \wedge m = m' \wedge \texttt{validTreeBase}(B, f, l, sub)$
if $cur' = \texttt{cond}(c, rl, lbl) :: l$ and $t = \texttt{cond}(c', rl', sub_1, sub_2)$:
return $c = c' \wedge lr = lr' \wedge \texttt{validTreeBase}(B, f, l, sub_2) \wedge$

if $lbl \in B$: return $\texttt{test_out}(sub1, lbl)$
else if $\texttt{find_label}$ (lbl,(fn_code f)) $= \lfloor l \rfloor'$: return $\texttt{validTreeBase}(B, f, l', sub_1)$
else return $\texttt{false}$
if $cur' = \texttt{label}(lbl) :: l$ and $t = out(lbl')$
return $lbl = lbl'$
if $cur' = goto(lbl) :: l$ and $t = out(lbl')$
return $lbl = lbl'$
in all other cases: false

4.2.2 Correctness and completeness of the conversion

The equivalence check $C \sim g$ defined above enjoys the desired semantic equivalence property:

Lemma 4.1. *Let p be a Mach program and p' a corresponding program where function bodies are represented as graphs of trees. Assume that $p(id).\texttt{code} \sim p'(id).\texttt{code}$ for all function names $id \in p$. Let G and G' be the global environments associated with p and p', respectively. If $F.\texttt{code} \sim F'.\texttt{code}$, then $G \vdash F, P, R, M \Rightarrow R', M'$ in the original Mach semantics if and only if $G' \vdash F', P, R, M \Rightarrow R', M'$ in the tree-based semantics of section 4.2.*

4.3 A validator for trace scheduling

We now present the validator for trace scheduling. The validator for trace scheduling is designed in two steps, as for the list scheduling validator: first, we validate pairs of trees ; then, function bodies.

4.3.1 Validation at the level of trees

We first define a validator V_t that checks semantic preservation between two instruction trees T_1, T_2.

$V_t(T_1, T_2, (m_1, s_1), (m_2, s_2)) =$
if $T_1 = \texttt{seq}(i_1, T_1')$:
return $V_t(T_1', T_2, \alpha(i_1, (m_1, s_1)), (m_2, s_2))$
if $T_2 = \texttt{seq}(i_2, T_2')$:
return $V_t(T_1, T_2', (m_1, s_1), \alpha(i_2, (m_2, s_2))$
if $T_1 = \texttt{cond}(cond_1, \vec{r}_1, T_1', T_1'')$ and $T_2 = \texttt{cond}(cond_2, \vec{r}_2, T_2', T_2'')$:
return $cond_1 = cond_2 \wedge m_1(\vec{r}_1) = m_2(\vec{r}_2)$
$\wedge V_t(T_1', T_2', (m_1, s_1), (m_2, s_2))$
$\wedge V_t(T_1'', T_2'', (m_1, s_1), (m_2, s_2))$
if $T_1 = \texttt{out}(l_1)$ and $T_2 = \texttt{out}(l_2)$:

return $l_2 = l_1 \wedge m_2 = m_1 \wedge s_2 \subseteq s_1$
in all other cases:
return `false`

The validator traverses the two trees in parallel, performing symbolic evaluation of the non-branching instructions. We reuse the $\alpha(i, (m, s))$ function of section 3.2.2. The (m_1, s_1) and (m_2, s_2) parameters are the current states of symbolic evaluation for T_1 and T_2, respectively. We process non-branching instructions repeatedly in T_1 or T_2 until we reach either two `cond` nodes or two `out` leaves. When we reach `cond` nodes in both trees, we check that the conditions being tested and the symbolic evaluations of their arguments are identical, so that at run-time control will flow on the same side of the conditional in both codes. We then continue validation on the `true` subtrees and on the `false` subtrees. Finally, when two `out` leaves are reached, we check that they branch to the same label and that the symbolic states agree ($m_2 = m_1$ and $s_2 \subseteq s_1$), as in the case of block verification.

4.3.2 Validation at the level of function bodies

We now extend the tree validator V_t to a validator that operates over two control-flow graphs of trees. We simply check that identically-labeled regular trees in both graphs are equivalent according to V_t, and that call trees and return trees are identical in both graphs.

$V_g(g_1, g_2) =$
if $\mathrm{Dom}(g_1) \neq \mathrm{Dom}(g_2)$, return `false`
for each $l \in \mathrm{Dom}(g_1)$:
if $g_1(l)$ and $g_2(l)$ are regular trees:
if $V_t(g_1(l), g_2(l), (\varepsilon, \emptyset), (\varepsilon, \emptyset)) =$ `false`, return `false`
otherwise:
if $g_1(l) \neq g_2(l)$, return `false`
end for each
return `true`

Before invoking the validator V_g, we need to convert the original and scheduled codes from the list-of-instructions representation to the graph-of-trees representation. The full validator for trace scheduling is therefore of the following form:

$V(F_1, F_2) =$
convert F_1.`code` to a graph of trees g_1
convert F_2.`code` to a graph of trees g_2
return F_1.`code` $\sim g_1 \wedge F_2$.`code` $\sim g_2 \wedge V_g(g_1, g_2)$

4.4 Proof of correctness

The proof follows the structure of the validator.

4.4.1 Correctness of the validation over trees

As expected, a successful run of V_t entails a semantic preservation result.

Lemma 4.2. *If* $V_t(T_1, T_2) = \texttt{true}$ *and* $\Sigma \vdash T_1, R, F, M \Rightarrow l, R', F', M'$ *then* $\Sigma \vdash T_2, R, F, M \Rightarrow l, R', F', M'$

Proof. To prove this lemma we use an operator $\texttt{plug}(c, t)$ which "plugs" the sequence of instructions c on top of the tree t. It is defined as $\texttt{plug}(i_1...i_n, t) = \texttt{seq}(i_1, (..., \texttt{seq}(i_n, t)...))$.
The theorem is a consequence of the stronger theorem:
If $\alpha(c_1) = (b_1, s_1)$ and $\alpha(c_2) = (b_2, s_2)$ and $V_t(T_1, T_2) = \texttt{true}$ and $\Sigma \vdash (c1; w), R_0, F_0, M_0 \Rightarrow w, R, F, M$ and $\Sigma \vdash T_1, R, F, M \Rightarrow l, R', F', M'$ then $\Sigma \vdash \texttt{plug}(c_2, T_2), R, F, M \Rightarrow R', F', M'$.
We prove this by functional induction: a case analysis and a proof by induction that follows the definition of the function. This is possible because the function is defined inductively on the sum of the heights of both trees.
Intuitively, the proof works by collecting all the instructions encountered during the executions of both trees, by proving that every time we encounter a condition, both executions flows in the same direction. When both executions reach a leaf, the two sequences of non-branching instructions, let us say c_1 and c_2, that have been collected are such that, with $\alpha(c_1) = (b_1, s_1)$ and $\alpha(c_2) = (b_2, s_2)$, $b_1 = b_2$ and $s_2 \subseteq s_1$. By using theorems 3.1 and 3.2, the result follows. □

4.4.2 Correctness of validation over function bodies

This validator is correct in the following sense. Let p, p' be two programs in the tree-based representation such that p' is identical to p except for the function bodies, and $V_g(p(id).\texttt{graph}, p'(id).\texttt{graph}) = \texttt{true}$ for all function names $id \in p$.

Theorem 4.1. *Let* G *and* G' *be the global environments associated with* p *and* p'*, respectively. If* $G \vdash F, P, R, M \Rightarrow_{traces} R', M'$ *and* $V_g(F.\texttt{graph}, F'.\texttt{graph}) = \texttt{true}$*, then* $G' \vdash F', P, R, M \Rightarrow_{traces} R', M'$.

The combination of Theorem 4.1 and Lemma 4.1 establishes the correctness of the validator for trace scheduling.

4.5 Discussion

We now discuss the implementation and evaluation of the validator we have presented.

4.5.1 Implementation

The validator has been implemented in the Coq proof-assistant version 8.1. It accounts for 2661 lines of Coq code and specifications and 5501 lines of proof. Table 4.1 is a detailed line count showing, for each component of the validators, the size of the specifications (*i.e.* the algorithms and the semantics) and the size of the proofs. The trace scheduling transformation has been implemented in OCaml. It took approximately six person-months to build the transformation, the validator and its proof.

Since Coq is a logic of total functions, the function definitions must be written in a so-called "structurally recursive" style where termination is obvious. All our validation functions are naturally structurally recursive, except validation between trees (function V_t in section 4.3.1) and validation of list-to-tree conversion (the function corresponding to the $\sim$ predicate in section 4.2.1).

For validation between trees, we used well-founded recursion, using the sum of the heights of the two trees as the decreasing, positive measure. Coq 8.1 provides good support for this style of recursive function definitions (the `Function` mechanism [BFPR06]) and for the corresponding inductive proof principles (the `functional induction` tactic).

Validation of list-to-tree conversion could fail to terminate if the original, list-based code contains a loop that does not cross any synchronization point. This indicates a bug in the external converter, since normally synchronization points include all targets of back edges. To detect this situation and to make the recursive definition of the validator acceptable to Coq, we add a counter as parameter to the validation function, initialized to the number of instructions in the original code and decremented every time we examine an instruction. If this counter drops to zero, validation stops on error.

It is interesting to note that the validation of trace scheduling is no larger and no more difficult than that of list scheduling. This is largely due to the use of the graph-of-trees representation of the code. However, the part labeled "tree semantics", which includes the definition and semantics of trees plus the validation of the conversion from list-of-instructions to graph-of-trees, is the largest and most difficult part of this development.

4.5.2 Experimental evaluation and complexity analysis

The verified compiler pass that we obtain has been integrated in the Compcert compiler (a version of the Compiler that is older than the one distributed) and tested on the test suite. All tests were successfully scheduled and validated after scheduling. Manual inspection of the scheduled code reveals that the schedulers performed a fair number of instruction reorderings and insertion of stubs. Validation was effective from a compiler engineering viewpoint: not only manual injection of errors in the schedulers were correctly caught, but the validator also found one unintentional bug in our first implementation of trace scheduling.

	Specifi-cations	Proofs	Total
Symbolic evaluation	736	1079	1815
Trace validation	234	1045	1279
Tree semantics	986	2418	3404
Trace scheduling validation	285	352	637
Label manipulation	306	458	764
Typing	114	149	263
Total	2661	5501	8162

Table 4.1: Size of the development (in non-blank lines of code, without comments)

To assess the compile-time overheads introduced by validation, we measured the execution times of the transformation and of the validator. Table 4.2 presents the results.

Test program	Trace scheduling		
	Transformation	Validation	Ratio V/T
`fib`	0.44 ms	0.58 ms	1.32
`integr`	1.0 ms	1.2 ms	1.15
`qsort`	1.8 ms	3.3 ms	1.89
`fft`	19 ms	62 ms	3.26
`sha1`	12 ms	24 ms	2.00
`aes`	67 ms	830 ms	12.25
`almabench`	56 ms	200 ms	3.57
`stopcopy`	4.9 ms	6.1 ms	1.25
`marksweep`	6.8 ms	11 ms	1.69

Table 4.2: Compilation times and verification times

As for list scheduling, the results show some inefficiencies for the AES case. It is due again to the potential explosion when comparing two symbolic expressions. There is also another inefficiency specific to trace scheduling. The tree-based representation of code that we use for validation can be exponentially larger than the original code represented as a list of instructions, because of tail duplication of basic blocks. This potential explosion caused by tail duplication can be avoided by adding more synchronization points: not just targets of back edges, but also some other labels chosen heuristically to limit tail duplication. For instance, we can mark as synchronization points all the labels that do not belong to the traces that the scheduler chose to optimize. Such heuristic choices are performed entirely in unverified code (the scheduler and the converter from list- to tree-based code representations) and have no impact on the validators and on their proofs of correctness.

4.5.3 Conclusion

The validator for trace scheduling we have presented shows that it is easy to extend the techniques of symbolic evaluation to the scope of regions and to handle control. This result relies on a change of representation and semantics of the program that encodes the important property that transformations are limited to regions. This change of representation turns out to be the most difficult part of the validator, even though we used translation validation to verify the full abstraction of the change of representation.

Even though the validator was designed with trace scheduling in mind, we believe that our validator is complete for transformations that reorder, factor out or duplicate instructions within traces, extended blocks or hyper blocks. The validator for trace scheduling is a generalization of the validator presented in the previous chapter. A question that arises is whether it is possible to generalize symbolic evaluation-based validation techniques to global programs or loops since the algorithm relies on the fact that the sequence of instructions is finite.

Chapter 5

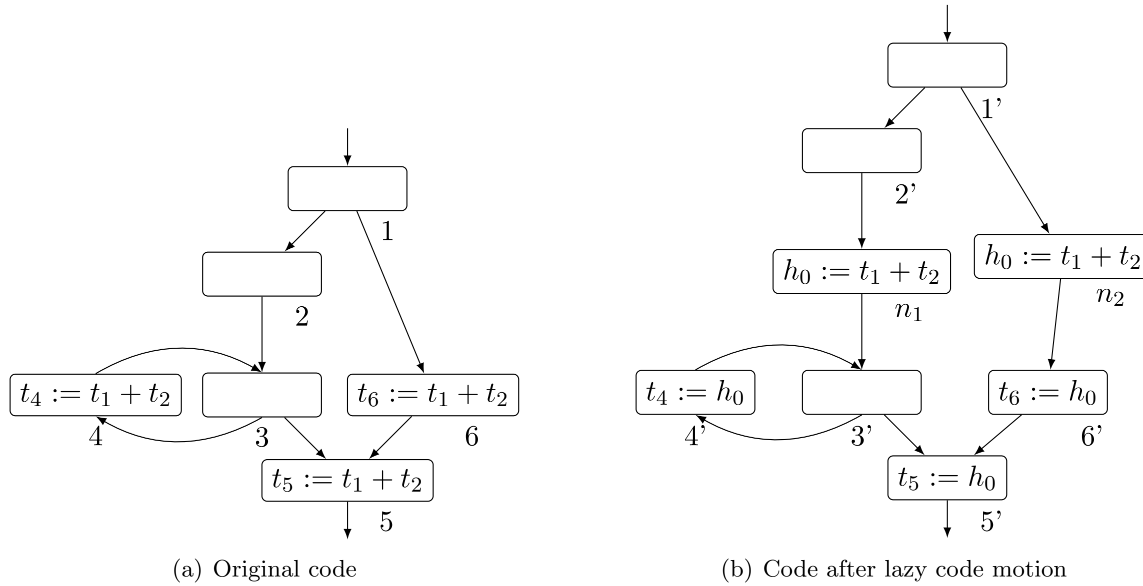
Lazy code motion

Many classic compiler optimizations exploit the results of dataflow analyses [Muc97]: constant propagation, partial redundancy elimination, register allocation... In this chapter, we present and formally verify a validator for an aggressive dataflow-based optimization: lazy code motion. Unlike the instruction scheduling optimizations of chapter 3 and 4, lazy code motion is a global optimization that moves instructions across basic blocks and even across loop boundaries. Its validation therefore requires new techniques that go beyond symbolic evaluation. Our validator is based on a dataflow analysis that is much simpler than the analyzes performed by the transformation. A technical problem that arises is to verify that diverging executions are not transformed into erroneous one. To address this problem we introduce the anticipability checking and prove its correctness by reasoning on the future of the execution of the original program.

5.1 Lazy code motion

Lazy code motion (LCM) [KRS92, KRS94] is a dataflow-based algorithm for the placement of computations within control flow graphs. It takes place at the RTL level. It suppresses unnecessary recomputations of values by moving their first computations earlier in the execution flow (if necessary), and later reusing the results of these first computations. Thus, LCM performs elimination of common subexpressions (both within and across basic blocks), as well as loop invariant code motion. In addition, it can also factor out partially redundant computations: computations that occur multiple times on some execution paths, but once or not at all on other paths. LCM is used in production compilers, for example in GCC version 4.

Figure 5.1 presents an example of lazy code motion. The original program in part (a) presents several interesting cases of redundancies for the computation of $t_1 + t_2$: loop invariance (node 4), simple straight-line redundancy (nodes 6 and 5), and partial redundancy (node 5). In the transformed program (part (b)), these redundant computations of $t_1 + t_2$ have all been



(a) Original code

(b) Code after lazy code motion

Figure 5.1: An example of lazy code motion transformation

eliminated: the expression is computed at most once on every possible execution path. Two instructions (node n_1 and n_2) have been added to the graph, both of which compute $t_1 + t_2$ and save its result into a fresh temporary h_0. The three occurrences of $t_1 + t_2$ in the original code have been rewritten into move instructions (nodes 4', 5' and 6'), copying register h_0 to the original destinations of the instructions.

The reader might wonder why two instructions $h_0 := t_1 + t_2$ were added in the two branches of the conditional, instead of a single instruction before node 1. The latter is what the partial redundancy elimination optimization of Morel and Renvoise [MR79] would do. However, this would create a long-lived temporary h_0, therefore increasing register pressure in the transformed code. The "lazy" aspect of LCM is that computations are placed as late as possible while avoiding repeated computations.

The LCM algorithm exploits the results of 4 dataflow analyses: up-safety (also called availability), down-safety (also called anticipability), delayability and isolation. These analyses can be implemented efficiently using bit vectors. Their results are then cleverly combined to determine an optimal placement for each computation performed by the initial program.

Knoop [KRS94] presents a correctness proof for LCM. However, mechanizing this proof appears difficult. Unlike the program transformations that have already been mechanically verified in the CompCert project, LCM is a highly non-local transformation: instructions are moved across basic blocks and even across loops. Moreover, the transformation generates fresh temporaries, which adds significant bureaucratic overhead to mechanized proofs. It appears easier to follow the verified validator approach. An additional benefit of this approach is that the LCM implementation can use efficient imperative data structures, since we do not need to

formally verify them. Moreover, it makes it easier to experiment with other variants of LCM.

To design and prove correct a translation validator for LCM, it is not important to know all the details of the analyses that indicate where new computations should be placed and which instructions should be rewritten. However it is important to know what kind of transformations happen. Two kinds of rewritings of the graph can occur:

- The nodes that exist in the original code (like node 4 in figure 5.1) still exist in the transformed code. The instruction they carry is either unchanged or can be rewritten as a move if they are arithmetic operations or loads (but not calls, tail calls, stores, returns nor conditions).
- Some fresh nodes are added (like node n_1) to the transformed graph. Their left-hand side is a fresh register; their right-hand side is the right-hand side of some instructions in the original code.

There exists an injective function from nodes of the original code to nodes of the transformed code. We call this mapping φ. It connects each node of the source code to its (possibly rewritten) counterpart in the transformed code. In the example of figure 5.1, φ maps nodes $1 \ldots 6$ to their primed versions $1' \ldots 6'$. We assume the unverified implementation of LCM is instrumented to produce this function. (In our implementation, we arrange that φ is always the identity function.) Nodes that are not in the image of φ are the fresh nodes introduced by LCM.

5.2 A validator for lazy code motion

In this section, we detail a translation validator for LCM.

5.2.1 General structure

Since LCM is an intraprocedural optimization, the validator proceeds function per function: each internal function F of the original program is matched against the identically-named function F' of the transformed program. Moreover, LCM does not change the type signature, parameter list and stack size of functions, and can be assumed not to change the entry point (by inserting `nop`s at the graph entrance if needed). Checking these invariants is easy; hence, we can focus on the validation of function graphs. Therefore, the validation algorithm is of the following shape:

$\texttt{validate}(F, F', \varphi) =$
  let $\mathcal{A}E = \texttt{analyze}(F')$ in
  $F'.\texttt{sig} = F.\texttt{sig}$ and $F'.\texttt{params} = F.\texttt{params}$ and
  $F'.\texttt{stack} = F.\texttt{stack}$ and $F'.\texttt{start} = F.\texttt{start}$ and
  for each node n of F, $\quad V(F, F', n, \varphi, \mathcal{A}E) = \texttt{true}$

As discussed in section 5.1, the φ parameter is the mapping from nodes of the input graph to nodes of the transformed graph provided by the implementation of LCM. The `analyze` function is a static analysis computing available expressions, described below in section 5.2.2. The V function validates pairs of matching nodes and is composed of two checks: `unify`, described in section 5.2.2 and `path`, described in section 5.2.3.

$$
\begin{aligned}
&V(F, F', n, \varphi, \mathcal{A}E) = \\
&\qquad \texttt{unify}(\mathcal{RD}(n'), F.\texttt{graph}(n), F'.\texttt{graph}(\varphi(n))) \\
&\quad \text{and for all successor } s \text{ of } n \text{ and matching successor } s' \text{ of } n', \\
&\qquad \texttt{path}(F.\texttt{graph}, F'.\texttt{graph}, s', \varphi(s))
\end{aligned}
$$

As outlined above, our implementation of a validator for LCM is carefully structured in two parts: a generic, rather bureaucratic framework parameterized over the `analyze` and V functions; and the LCM-specific, more subtle functions `analyze` and V. As we will see in this chapter, this structure facilitates the correctness proof of the validator. It also makes it possible to reuse the generic framework and its proof in other contexts, as illustrated in section 5.4.

We now focus on the construction of V, the node-level validator, and the static analysis it exploits.

5.2.2 Verification of the equivalence of single instructions

Consider an instruction i at node n in the original code and the corresponding instruction i' at node $\varphi(n)$ in the code after LCM (for example, nodes 4 and $4'$ in figure 5.1). We wish to check that these two instructions are semantically equivalent. If the transformation was a correct LCM, two cases are possible:

- $i = i'$: both instructions will obviously lead to equivalent run-time states, if executed in equivalent initial states.
- i' is of the form $r := h$ for some register r and fresh register h, and i is of the form $r := \mathit{rhs}$ for some right-hand side rhs, which can be either an arithmetic operation `op`$(\ldots)$ or a memory read `load`$(\ldots)$.

In the latter case, we need to verify that rhs and h produce the same value. More precisely, we need to verify that the value contained in h in the transformed code is equal to the value produced by evaluating rhs in the original code. LCM being a purely syntactical redundancy elimination transformation, it must be the case that the instruction $h := \mathit{rhs}$ exists on every path leading to $\varphi(n)$ in the transformed code; moreover, the values of h and rhs are preserved along these paths. This property can be checked by performing an available expression analysis on the transformed code.

$$
\begin{array}{lcl}
T(\mathtt{nop}(s), \mathcal{E}) & = & \mathcal{E} \\
T(\mathtt{op}(op, \vec{r}, r, s), \mathcal{E}) & = & \mathcal{E} \cup \{r = \mathtt{op}(op, \vec{r})\} \setminus \{\text{equalities reading } r\} \\
T(\mathtt{load}(\kappa, mode, \vec{r}, r, s), \mathcal{E}) & = & \mathcal{E} \cup \{r = \mathtt{load}(\kappa, mode, \vec{r})\} \setminus \{\text{equalities reading } r\} \\
T(\mathtt{store}(chunk, addr, \vec{r}, src, s), \mathcal{E}) & = & \mathcal{E} \setminus \{\text{equalities involving a } \mathtt{load}\} \\
T(\mathtt{call}(sig, ros, args, res, s), \mathcal{E}) & = & \mathcal{E} \setminus \{\text{equalities reading } r \text{ or involving } \mathtt{load}\} \\
T(\mathtt{tailcall}(sig, ros, args), \mathcal{E}) & = & \mathcal{E} \\
T(\mathtt{cond}(cond, args, ifso, ifnot), \mathcal{E}) & = & \mathcal{E} \\
T(\mathtt{return}(optarg), \mathcal{E}) & = & \mathcal{E}
\end{array}
$$

Figure 5.2: Transfer function for available expressions

Available expressions

The available expression analysis produces, for each program point of the transformed code, a set of equations $r = rhs$ between registers and right-hand sides. (For efficiency, we encode these sets as finite maps from registers to right-hand sides, represented as Patricia trees.) Available expressions is a standard forward dataflow analysis:

$$\mathcal{A}E(s) = \bigcap \{T(F'.\mathtt{graph}(l), \mathcal{A}E(l)) \mid s \text{ is a successor of } l\}$$

The join operation is set intersection; the top element of the lattice is the empty set, and the bottom element is a symbolic constant $\mathcal{U}$ denoting the universe of all equations. The transfer function T is standard; more details can be found in figure 5.2. For instance, if the instruction i is the operation $r := t_1 + t_2$, and R is the set of equations "before" i, the set $T(i, R)$ of equations "after" i is obtained by adding the equality $r = t_1 + t_2$ to R, then removing every equality in this set that uses register r (including the one just added if t_1 or t_2 equals r). We also track equalities between register and `load` instructions. Those equalities are erased whenever a `store` instruction is encountered because we do not maintain aliasing information.

To solve the dataflow equations, we reuse the generic implementation of Kildall's algorithm provided by the CompCert compiler. Leveraging the correctness proof of this solver and the definition of the transfer function, we obtain that the equations inferred by the analysis hold in any concrete execution of the transformed code. For example, if the set of equations at point l include the equality $r = t_1 + t_2$, it must be the case that $R(r) = R(t_1) + R(t_2)$ for every possible execution of the program that reaches point l with a register state R.

Instruction unification

Armed with the results of the available expression analysis, the `unify` check between pairs of matching instructions can be easily expressed:

$\mathtt{unify}(D, i, i') =$

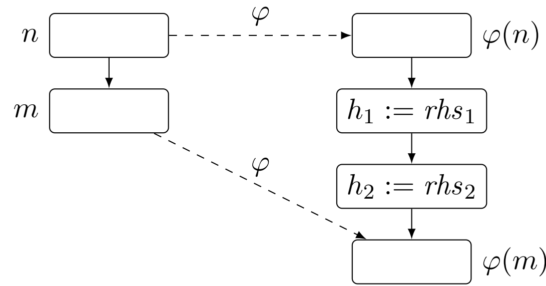


Figure 5.3: Effect of the transformation on the structure of the code

if $i' = i$ then `true` else
case (i, i') of
| $(r := \texttt{op}(op, \vec{r}),\ r := h) \rightarrow (h = \texttt{op}(op, \vec{r})) \in D$
| $(r := \texttt{load}(\kappa, mode, \vec{r}),\ r := h) \rightarrow (h = \texttt{load}(\kappa, mode, \vec{r})) \in D$
| otherwise $\rightarrow$ `false`

Here, $D = \mathcal{A}E(n')$ is the set of available expressions at the point n' where the transformed instruction i' occurs. Either the original instruction i and the transformed instruction i' are equal, or the former is $r := rhs$ and the latter is $r := h$, in which case instruction unification succeeds if and only if the equation $h = rhs$ is known to hold according to the results of the available expression analysis.

5.2.3 Verifying the flow of control

Unifying pairs of instructions is not enough to guarantee semantic preservation: we also need to check that the control flow is preserved. For example, in the code shown in figure 5.1, after checking that the conditional tests at nodes 1 and $1'$ are identical, we must make sure that whenever the original code transitions from node 1 to node 6, the transformed code can

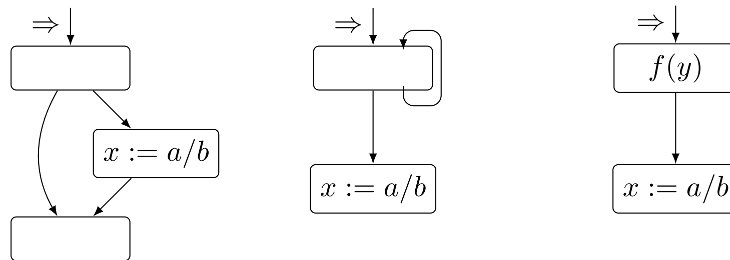


Figure 5.4: Three examples of incorrect code motion. Placing a computation of a/b at the program points marked by $\Rightarrow$ can potentially transform a well-defined execution into an erroneous one.

transition from node $1'$ to $6'$, executing the anticipated computation at n_2 on its way.

More generally, if the k-th successor of n in the original CFG is m, there must exist a path in the transformed CFG from $\varphi(n)$ to $\varphi(m)$ that goes through the k-th successor of $\varphi(n)$. (See figure 5.3.) Since instructions can be added to the transformed graph during lazy code motion, $\varphi(m)$ is not necessarily the k-th successor of $\varphi(n)$: one or several anticipated computations of the shape $h := rhs$ may need to be executed. Here comes a delicate aspect of our validator: not only there must exist a path from $\varphi(n)$ to $\varphi(m)$, but moreover the anticipated computations $h := rhs$ found on this path must be semantically well-defined: they should not go wrong at run-time. This is required to ensure that whenever an execution of the original code transitions in one step from n to m, the transformed code can transition (possibly in several steps) from $\varphi(n)$ to $\varphi(m)$ without going wrong.

Figure 5.4 shows three examples of code motion where this property may not hold. In all three cases, we consider anticipating the computation a/b (an integer division that can go wrong if $b = 0$) at the program points marked by a double arrow. In the leftmost example, it is obviously unsafe to compute a/b before the conditional test: quite possibly, the test in the original code checks that $b \neq 0$ before computing a/b. The middle example is more subtle: it could be the case that the loop preceding the computation of a/b does not terminate whenever $b = 0$. In this case, the original code never crashes on a division by zero, but anticipating the division before the loop could cause the transformed program to do so. The rightmost example is similar to the middle one, with the loop being replaced by a function call. The situation is similar because the function call may not terminate when $b = 0$.

How, then, can we check that the instructions that have been added to the graph are semantically well-defined? Because we distinguish erroneous executions and diverging executions, we cannot rely on a standard anticipability analysis. Our approach is the following: whenever we encounter an instruction $h := rhs$ that was inserted by the LCM transformation on the path from $\varphi(n)$ to $\varphi(m)$, we check that the computation of rhs is *inevitable* in the original code starting at node m. In other words, all execution paths starting from m in the original code must, in a finite number of steps, compute rhs. Since the semantic preservation result that we wish to establish takes as an assumption that the execution of the original code does not go wrong, we know that the computation of rhs cannot go wrong, and therefore it is legal to anticipate it in the transformed code. We now define precisely an algorithm, called the *anticipability checker*, that performs this check.

Anticipability checking

Our algorithm is described in figure 5.5. It takes four arguments: a graph g, an instruction right-hand side rhs to search for, a program point l where the search begins and a map S that associates to every node a marker. Its goal is to verify that on every path starting at l in

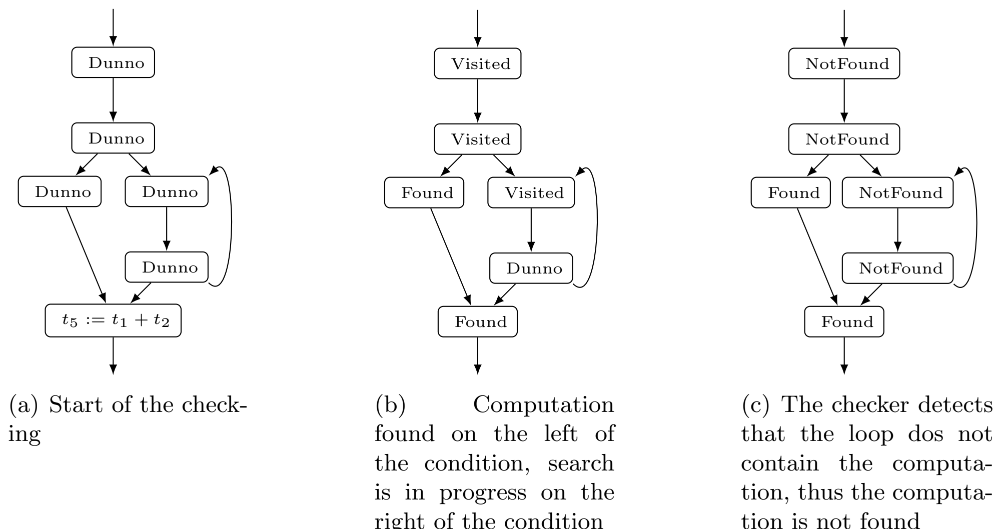
```
1   function ant_checker_rec (g,rhs,p_c,S) =

3       case S(p_c) of
        | Found →(S,true)
5       | NotFound →(S,false)
        | Visited →(S,false)
7       | Dunno →

9           case g(p_c) of
            | return _ →(S{p_c ←NotFound},false)
11          | tailcall (_,_,_) →(S{p_c ←NotFound},false)
            | cond (_,_,l_true,l_false) →
13                  let (S',b_1) = ant_checker_rec (g,rhs,l_true,S{p_c ← Visited}) in
                    let (S'',b_2) = ant_checker_rec (g,rhs,l_false,S') in
15                  if b_1&& b_2then (S''{p_c ←Found},true) else (S''{p_c ←NotFound},false)
            | nop l→
17                  let (S',b) := ant_checker_rec (g,rhs,l,S{p_c ← Visited}) in
                    if b then (S'{p_c ←Found},true) else (S'{p_c ←NotFound},false)
19          | call (_,_,_,_,l) →(S{p_c ←NotFound},false)
            | store (_,_,_,_,l) →
21                  if rhs reads memory then (S{p_c ←NotFound},false) else
                    let (S',b) := ant_checker_rec (g,rhs,l,S{p_c ← Visited}) in
23                  if b then (S'{p_c ←Found},true) else (S'{p_c ←NotFound},false)
            | op (op,args,r,l) →
25                  if r is an operand of rhs then (S{p_c ←NotFound},false) else
                    if rhs = (op op args) then (S{p_c ←Found},true) else
27                      let (S',b) = ant_checker_rec (g,rhs,l,S{p_c ← Visited}) in
                        if b then (S'{p_c ←Found},true) else (S'{p_c ←NotFound},false)
29          | load (chk,addr,args,r,l) →
                    if r is an operand of rhs then (S{p_c ←NotFound},false) else
31                  if rhs = (load chk addr args) then (S{p_c ←Found},true) else
                        let (S',b) = ant_checker_rec (g,rhs,l,S{p_c ← Visited}) in
33                      if b then (S'{p_c ←Found},true) else (S'{p_c ←NotFound},false)


35
    function ant_checker (g,rhs,p_c) = let (S,b) = ant_checker_rec(g,rhs,p_c,(l ↦Dunno)) in b
```

Figure 5.5: Anticipability checker



(a) Start of the checking

(b) Computation found on the left of the condition, search is in progress on the right of the condition

(c) The checker detects that the loop dos not contain the computation, thus the computation is not found

Figure 5.6: A few steps of the anticipability checking for computation $t_1 + t_2$. (Only the node at the way out of the graph holds the computation.)

the graph g, execution reaches an instruction with right-hand side *rhs* such that none of the operands of *rhs* have been redefined on the path. Basically it is a depth-first search that covers all the path starting at l. Note that if there is a path starting at l that contains a loop so that *rhs* is neither between l and the loop nor in the loop itself, then there exists a path on which *rhs* is not reachable and that corresponds to an infinite execution. To obtain an efficient algorithm, we need to ensure that we do not go through loops several times. To this end, if the search reaches a join point not for the first time and where *rhs* was not found before, we must stop searching immediately. This is achieved through the use of four different markers over nodes:

- `Found` means that *rhs* is computed on every path from the current node.
- `NotFound` means that there exists a path from the current node in which *rhs* is not computed.
- `Dunno` is the initial state of every node before it has been visited.
- `Visited` is the state when a state is visited and we do not know yet whether *rhs* is computed on all paths or not. It is used to detect loops.

Let us detail a few cases. When the search reaches a node that is marked `Visited` (line 6), it means that the search went through a loop and *rhs* was not found. This could lead to a semantics discrepancy (recall the middle example in figure 5.4) and the search fails. For similar reasons, it also fails when a call is reached (line 19). When the search reaches an operation (line 24), we first verify (line 25) that r, the destination register of the instruction does not modify

the operands of *rhs*. Then, (line 26) if the instruction right-hand side we reached correspond to *rhs*, we found *rhs* and we mark the node accordingly. Otherwise, the search continues (line 27) and we mark the node based on whether the recursive search found *rhs* or not (line 28). Figure 5.6 gives an excerpt of an anticipability checking.

The `ant_checker` function, when it returns `Found`, should imply that the right-hand side expression is well defined. We prove that this is the case in section 5.3.5 below.

Verifying the existence of semantics paths

Once we can decide the well-definedness of instructions, checking for the existence of a path between two nodes of the transformed graph is simple. The function `path`(g, g', n, m) checks that there exists a path in CFG g' from node n to node m, composed of zero, one or several single-successor instructions of the form $h := rhs$. The destination register h must be fresh (unused in g) so as to preserve the abstract semantics equivalence invariant. Moreover, the right-hand side *rhs* must be safely anticipable: it must be the case that `ant_checker`$(g, rhs, \varphi^{-1}(m)) =$ `Found`, so that *rhs* can be computed before reaching m without getting stuck. The pseudocode of the `path` function is given below.

let `path` $(g, g', n, m) =$
  let rec `path_aux` $x =$
    if $x = m$: return `true`
    otherwise, if $g'.x$ is $r := rhs$ with successor y:
      return `ant_checker` $(g, rhs, \varphi^{-1}(n)) \wedge$ `path_aux` y
  in
`path_aux` n

5.3 Proof of correctness

Let P_i be an input program and P_o be the output program produced by the untrusted implementation of LCM. We wish to prove that if the validator succeeds on all pairs of matching functions from P_i and P_o, then $P_i \Downarrow B \Rightarrow P_o \Downarrow B$. In other words, if P_i does not go wrong and executes with observable behavior B, then so does P_o.

5.3.1 Simulating executions

The way we build a semantics preservation proof is to construct a relation between execution states of the input and output programs, written $S_i \sim S_o$, and show that it is a simulation:

- Initial states: if S_i and S_o are two initial states, then $S_i \sim S_o$.
- Final states: if $S_i \sim S_o$ and S_i is a final state, then S_o must be a final state.

- Simulation property: if $S_i \sim S_o$, any transition from state S_i with trace t is simulated by one or several transitions starting in state S_o, producing the same trace t, and preserving the simulation relation $\sim$.

The hypothesis that the input program P_i does not go wrong plays a crucial role in our semantic preservation proof, in particular to show the correctness of the anticipability criterion. Therefore, we reflect this hypothesis in the precise statement of the simulation property above, as follows. (G_i, G_o are the global environments corresponding to programs P_i and P_o, respectively.)

Definition 5.1 (Simulation property).
Let I_i be the initial state of program P_i and I_o that of program P_o. Assume that

- $S_i \sim S_o$ *(current states are related)*
- $G_i \vdash S_i \xrightarrow{t} S'_i$ *(the input program makes a transition)*
- $G_i \vdash I_i \xrightarrow{t'}{}^{*} S_i$ *and* $G_o \vdash I_o \xrightarrow{t'}{}^{*} S_o$ *(current states are reachable from initial states)*
- $G_i \vdash S'_i \Downarrow B$ *for some behavior B (the input program does not go wrong after the transition).*

Then, there exists S'_o such that $G_o \vdash S_o \xrightarrow{t}{}^{+} S'_o$ and $S'_i \sim S'_o$.

The commuting diagram corresponding to this definition is depicted below. Solid lines represent hypotheses; dashed lines represent conclusions.

$$
\begin{array}{llcl}
\text{Input program:} & I_i \xrightarrow[*]{t'} S_i & \xrightarrow{t} & S'_i \xrightarrow[\text{go wrong}]{\text{does not}} \\
 & \qquad\quad \sim | & & \ \ \vdots \sim \\
\text{Output program:} & I_o \xrightarrow[*]{t'} S_o & \dashrightarrow_{+}^{t} & S'_o
\end{array}
$$

It is easy to show that the simulation property implies semantic preservation:

Theorem 5.1. *Under the hypotheses between initial states and final states and the simulation property, $P_i \Downarrow B$ implies $P_o \Downarrow B$.*

5.3.2 The invariant of semantic preservation

We now construct the relation $\sim$ between execution states before and after LCM that acts as the invariant in our proof of semantic preservation. We first define a relation between register files.

Definition 5.2 (Equivalence of register files).
$f \vdash R \sim R'$ if and only if $R(r) = R'(r)$ for every register r that appears in an instruction of f's code.

This definition allows the register file R' of the transformed function to bind additional registers not present in the original function, especially the temporary registers introduced during LCM optimization. Equivalence between execution states is then defined by the three rules below.

Definition 5.3 (Equivalence of execution states)**.**

$$\frac{\texttt{validate}(f, f', \varphi) = \texttt{true} \qquad f \vdash R \sim R' \qquad G, G' \vdash \Sigma \sim_{\mathcal{F}} \Sigma'}{G, G' \vdash \mathcal{S}(\Sigma, f, \sigma, l, R, M) \sim \mathcal{S}(\Sigma', f', \sigma, \varphi(l), R', M)}$$

$$\frac{\mathcal{T}_{\mathcal{V}}(Fd) = Fd' \qquad G, G' \vdash \Sigma \sim_{\mathcal{F}} \Sigma'}{G, G' \vdash \mathcal{C}(\Sigma, Fd, \vec{v}, M) \sim \mathcal{C}(\Sigma', Fd', \vec{v}, M)}$$

$$\frac{G, G' \vdash \Sigma \sim_{\mathcal{F}} \Sigma'}{G, G' \vdash \mathcal{R}(\Sigma, v, M) \sim \mathcal{R}(\Sigma', v, M)}$$

Generally speaking, equivalent states must have exactly the same memory states and the same value components (stack pointer σ, arguments and results of function calls). As mentioned before, the register files R, R' of regular states may differ on temporary registers but must be related by the $f \vdash R \sim R'$ relation. The function parts f, f' must be related by a successful run of validation. The program points l, l' must be related by $l' = \varphi(l)$.

The most delicate part of the definition is the equivalence between call stacks $G, G' \vdash \Sigma \sim_{\mathcal{F}} \Sigma'$. The frames of the two stacks Σ and Σ' must be related pairwise by the following predicate.

Definition 5.4 (Equivalence of stack frames)**.**

$$\frac{\begin{array}{l} \qquad\qquad\qquad \texttt{validate}(f, f', \varphi) = \texttt{true} \qquad f \vdash R \sim R' \\ \forall v, M, B, \quad G \vdash \mathcal{S}(\Sigma, f, \sigma, l, R\{r \leftarrow v\}, M) \Downarrow B \\ \qquad \Longrightarrow \exists R'', \ f \vdash R\{r \leftarrow v\} \sim R'' \\ \qquad\qquad \wedge \ G' \vdash \mathcal{S}(\Sigma, f', \sigma, l', R'\{r \leftarrow v\}, M) \xrightarrow{\varepsilon}{}^{+} \mathcal{S}(\Sigma, f', \sigma, \varphi(l), R'', M) \end{array}}{G, G' \vdash \mathcal{F}(r, f, \sigma, l, R) \sim_{\mathcal{F}} \mathcal{F}(r, f', \sigma, l', R')}$$

The scary-looking third premise of the definition above captures the following condition: if we suppose that the execution of the initial program is well-defined once control returns to node l of the caller, then it should be possible to perform an execution in the transformed graph from l' down to $\varphi(l)$. This requirement is a consequence of the anticipability problem. As explained earlier, we need to make sure that execution is well defined from l' to $\varphi(l)$. But when the instruction is a function call, we have to store this information in the equivalence of frames, universally quantified on the not-yet-known return value v and memory state M at return time.

At the time we store the property we do not know yet if the execution will be semantically correct from l, so we suppose it until we get the information (that is, when execution reaches l).

Having stated semantics preservation as a simulation diagram and defined the invariant of the simulation, we now turn to the proof itself. We now give a high-level overview of the correctness proof for our validator. Besides giving an idea of how we prove the validation kernel (this proof differs from earlier paper proofs mainly on the handling of semantic well-definedness), we try to show that the burden of the proof can be reduced by adequate design.

5.3.3 A little bit of proof design

Recall that the validator is composed of two parts: first, a generic validator that requires an implementation of V and of `analyze`; second, an implementation of V and `analyze` specialized for LCM. The proof follows this structure: on one hand, we prove that if V satisfies the simulation property, then the generic validator implies semantics preservation; on the other hand, we prove that the node-level validation specialized for LCM satisfies the simulation property.

This decomposition of the proof improves re-usability and, above all, greatly improves abstraction for the proof that V satisfies the simulation property (which is the kernel of the proof on which we want to focus) and hence reduces the proof burden of the formalization. Indeed, many details of the formalization can be hidden in the proof of the framework. This includes, among other things, function invocation, function return, global variables, and stack management.

Besides, this allows us to prove that V only satisfies a weaker version of the simulation property that we call the validation property, and whose equivalence predicate is a simplification of the equivalence presented in section 5.3.2. In the simplified equivalence predicate, there is no mention of stack equivalence, function transformation, stack pointers or results of the validation.

Definition 5.5 (Abstract equivalence of states)**.**

$$\frac{f \vdash R \sim R' \qquad l' = \varphi(l)}{G, G' \vdash \mathcal{S}(\Sigma, f, \sigma, l, R, M) \approx_{\mathcal{S}} \mathcal{S}(\Sigma', f', \sigma, l', R', M)}$$

$$G, G' \vdash \mathcal{C}(\Sigma, Fd, \vec{v}, M) \approx_{\mathcal{C}} \mathcal{C}(\Sigma', Fd', \vec{v}, M)$$

$$G, G' \vdash \mathcal{R}(\Sigma, v, M) \approx_{\mathcal{R}} \mathcal{R}(\Sigma', v, M)$$

The validation property is stated in three version, one for regular states, one for calls and one for return. We present only the property for regular states. If $S = \mathcal{S}(\Sigma, f, \sigma, l, R, M)$ is a regular state, we write $S.f$ for the f component of the state and $S.l$ for the l component.

Definition 5.6 (Validation property)**.**
Let I_i be the initial state of program P_i and I_o that of program P_o. Assume that

- $S_i \approx_{\mathcal{S}} S_o$
- $G_i \vdash S_i \xrightarrow{t} S_i'$
- $G_i \vdash I_i \xrightarrow{t'}{}^{*} S_i$ *and* $G_o \vdash I_o \xrightarrow{t'}{}^{*} S_o$
- $S_i' \Downarrow B$ *for some behavior* B
- $V(S_i.f, S_o.f, S_i.l, \varphi, \texttt{analyze}(S_o.f)) = \texttt{true}$

Then, there exists S_o' *such that* $S_o \xrightarrow{t}{}^{+} S_o'$ *and* $S_i' \approx S_o'$*.*

We then prove that if V satisfies the validation property, and if the two programs P_i, P_o successfully pass validation, then the simulation property (definition 5.1) is satisfied, and therefore (theorem 5.1) semantic preservation holds. This proof is not particularly interesting but represents a large part of the Coq development and requires a fair knowledge of CompCert internals.

We now outline the formal proof of the fact that V satisfies the validation property, which is the more interesting part of the proof.

5.3.4 Verification of the equivalence of single instructions

We first need to prove the correctness of the available expression analysis. The predicate $S \models \mathcal{E}$ states that a set of equalities $\mathcal{E}$ inferred by the analysis are satisfied in execution state S. The predicate is always true on call states and on return states.

Definition 5.7 (Correctness of a set of equalities)**.**
$\mathcal{S}(\Sigma, f, \sigma, l, R, M) \models \mathcal{RD}(l)$ *if and only if*

- $(r = \texttt{op}(op, \vec{r})) \in \mathcal{RD}(l)$ *implies* $R(r) = \texttt{eval_op}(op, R(\vec{r}))$
- $(r = \texttt{load}(\kappa, mode, \vec{r})) \in \mathcal{RD}(l)$ *implies*
  $\texttt{eval_addressing}(mode, \vec{r}) = v$ *and* $R(r) = \texttt{load}(\kappa, v)$ *for some pointer value* v*.*

The correctness of the analysis can now be stated:

Lemma 5.1 (Correctness of available expression analysis)**.** *Let* S^0 *be the initial state of the program. For all regular states* S *such that* $S^0 \xrightarrow{t}{}^{*} S$*, we have* $S \models \texttt{analyze}(S.f)$*.*

Then, it is easy to prove the correctness of the unification check. The predicate $\approx_{\mathcal{S}}^{W}$ is a weaker version of $\approx_{\mathcal{S}}$, where we remove the requirement that $l' = \varphi(l)$, therefore enabling the program counter of the transformed code to temporarily get out of synchronization with that of the original code.

Lemma 5.2. *Assume*

- $S_i \approx_{\mathcal{S}} S_o$
- $S_i \xrightarrow{t} S'_i$
- $\texttt{unify}(\texttt{analyze}(S_o.f), S_i.f.\texttt{graph}, S_o.f.\texttt{graph}, S_i.l, S_o.l) = \texttt{true}$
- $I_o \xrightarrow{t'}{}^* S_o$

Then, there exists a state S''_o such that $S_o \xrightarrow{t} S''_o$ and $S'_i \approx^W_{\mathcal{S}} S''_o$

Indeed, from the hypothesis $I_o \xrightarrow{t}{}^* S_o$ and the correctness of the analysis, we deduce that $S_o \models \texttt{analyze}(S_o.f)$, which implies that the equality used during the unification, if any, holds at run-time. This illustrate the use of hypothesis on the past of the execution of the transformed program. By doing so, we avoid to maintain the correctness of the analysis in the predicate of equivalence.

It remains to step through the transformed CFG, as performed by path checking, in order to go from the weak abstract equivalence $\approx^W_{\mathcal{S}}$ to the full abstract equivalence $\approx_{\mathcal{S}}$.

5.3.5 Anticipability checking

Before proving the properties of path checking, we need to prove the correctness of the anticipability check: if the check succeeds and the semantics of the input program is well defined, then the right-hand side expression given to the anticipability check is well defined.

Lemma 5.3. *Assume* $\texttt{ant_checker}(f.graph, rhs, l) = \texttt{true}$ *and* $\mathcal{S}(\Sigma, f, \sigma, l, R, M) \Downarrow B$ *for some* B*. Then, there exists a value* v *such that* rhs *evaluates to* v *(without run-time errors) in the state* R, M*.*

Then, the semantic property guaranteed by path checking is that there exists a sequence of reductions from $\texttt{successor}(\varphi(n))$ to $\varphi(\texttt{successor}(n))$ such that the abstract invariant of semantic equivalence is reinstated at the end of the sequence.

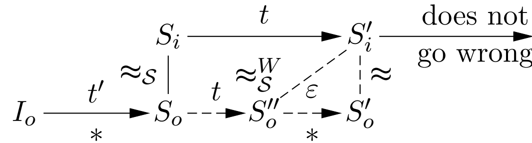
Lemma 5.4. *Assume*

- $S'_i \approx^W_{\mathcal{S}} S''_o$
- $\texttt{path}(S'_i.f.\texttt{graph}, S''_o.f.\texttt{graph}, S''_o.l, \varphi(S_i.l)) = \texttt{true}$
- $S'_i \Downarrow B$ *for some* B

Then, there exists a state S'_o such that $S''_o \xrightarrow{\varepsilon}{}^* S'_o$ and $S'_i \approx S'_o$

This illustrates the use of the hypothesis on the future of the execution of the initial program. All the proofs are rather straightforward once we know that we need to reason on the future of the execution of the initial program.

By combining lemmas 5.2 and 5.4 we prove the validation property for regular states, according to the following diagram.



The proofs of the validation property for call and return states are similar.

5.4 Discussion

5.4.1 Implementation

The LCM validator and its proof of correctness were implemented in the Coq proof assistant. The Coq development is approximately 5000 lines long. 800 lines correspond to the specification of the LCM validator, in pure functional style, from which executable Caml code is automatically generated by Coq's extraction facility. The remaining 4200 lines correspond to the correctness proof. In addition, a lazy code motion optimization was implemented in OCaml, in roughly 800 lines of code.

The following table shows the relative sizes of the various parts of the Coq development.

Part	Size
General framework	37%
Anticipability check	16%
Path verification	7%
Reaching definition analysis	18%
Instruction unification	6%
Validation function	16%

Table 5.1: Size of the development

As discussed below, large parts of this development are not specific to LCM and can be reused: the general framework of section 5.3.3, anticipability checking, available expressions, etc. Assuming these parts are available as part of a toolkit, building and proving correct the LCM validator would require only 1100 lines of code and proofs.

5.4.2 Experimental evaluation and complexity analysis

Let N be the number of nodes in the initial CFG g. The number of nodes in the transformed graph g' is in $\mathcal{O}(N)$. We first perform an available expression analysis on the transformed graph, which takes time $\mathcal{O}(N^3)$. Then, for each node of the initial graph we perform an unification and a path checking. Unification is done in constant time and path checking tries to find a non-cyclic path in the transformed graph, performing an anticipability checking in time $\mathcal{O}(N)$

for instructions that may be ill-defined. Hence path checking is in $\mathcal{O}(N^2)$ but this is a rough pessimistic approximation.

In conclusion, our validator runs in time $\mathcal{O}(N^3)$. Since lazy code motion itself performs four data-flow analysis that run in time $\mathcal{O}(N^3)$, running the validator does not change the complexity of the lazy code motion compiler pass.

In practice, on our benchmark suite, the time needed to validate a function is on average 22.5% of the time it takes to perform LCM.

5.4.3 Completeness

We proved the correctness of the validator. This is an important property, but not sufficient in practice: a validator that rejects every possible transformation is definitely correct but also quite useless. We need evidence that the validator is relatively complete with respect to “reasonable” implementations of LCM. Formally specifying and proving such a relative completeness result is difficult, so we reverted to experimentation. We ran LCM and its validator on the CompCert benchmark suite (17 small to medium-size C programs) and on a number of examples hand-crafted to exercise the LCM optimization. No false alarms were reported by the validator.

More generally, there are two main sources of possible incompleteness in our validator. First, the external implementation of LCM could take advantage of equalities between right-hand sides of computations that our available expression analysis is unable to capture, causing instruction unification to fail. We believe this never happens as long as the available expression analysis used by the validator is identical to (or at least no coarser than) the up-safety analysis used in the implementation of LCM, which is the case in our implementation.

The second potential source of false alarms is the anticipability check. Recall that the validator prohibits anticipating a computation that can fail at run-time before a loop or function call. The CompCert semantics for the RTL language errs on the side of caution and treats all undefined behaviors as run-time failures: not just behaviors such as integer division by zero or memory loads from incorrect pointers, which can actually cause the program to crash when run on a real processor, but also behaviors such as adding two pointers or shifting an integer by more than 32 bits, which are not specified in RTL but would not crash the program during actual execution. (However, arithmetic overflows and underflows are correctly modeled as not causing run-time errors, because the RTL language uses modulo integer arithmetic and IEEE float arithmetic.) Because the RTL semantics treats all undefined behaviors as potential run-time errors, our validator restricts the points where e.g. an addition or a shift can be anticipated, while the external implementation of LCM could (rightly) consider that such a computation is safe and can be placed anywhere. This situation happened once in our tests.

One way to address this issue is to increase the number of operations that cannot fail in the RTL semantics. We could exploit the results of a simple static analysis that keeps track of the

shape of values (integers, pointers or floats), such as the trivial "int or float" type system for RTL used in [Ler08]. Additionally, we could refine the semantics of RTL to distinguish between undefined operations that can crash the program (such as loads from invalid addresses) and undefined operations that cannot (such as adding two pointers); the latter would be modeled as succeding, but returning an unspecified result. In both approaches, we increase the number of arithmetic instructions that can be anticipated freely.

5.4.4 Reusing the development

One advantage of translation validation is the re-usability of the approach. It makes it easy to experiment with variants of a transformation, for example by using a different set of data-flow analyzes in lazy code motion. It also happens that, in one compiler, two different versions of a transformation coexist. It is the case with `GCC`: depending on whether one optimizes for space or for time, the compiler performs partial redundancy elimination [MR79] or lazy code motion. We believe, without any formal proof, that the validator presented here works equally well for partial redundancy elimination. In such a configuration, the formalization burden is greatly reduced by using translation validation instead of compiler proof.

Classical redundancy elimination algorithms make the safe restriction that a computation e cannot be placed on some control flow path that does not compute e in the original program. As a consequence, code motion can be blocked by *preventing regions* [BGS98], resulting in less redundancy elimination than expected, especially in loops. A solution to this problem is *safe* speculative code motion [BGS98] where we lift the restriction for some computation e as long as e cannot cause run-time errors. Our validator can easily handle this case: the anticipability check is not needed if the new instruction is safe, as can easily be checked by examination of this instruction. Another solution is to perform control flow restructuring [Ste96, BGS98] to separate paths depending on whether they contain the computation e or not. This control flow transformation is not allowed by our validator and constitutes an interesting direction for future work.

To show that re-usability can go one step further, we have modified the unification rules of our lazy code motion validator to build a certified compiler pass of constant propagation with strength reduction. For this transformation, the available expression analysis needs to be performed not on the transformed code but on the initial one. Thankfully, the framework is designed to allow analyses on both programs. The modification mainly consists of replacing the unification rules for operation and loads, which represent about 3% of the complete development of LCM. (Note however that unification rules in the case of constant propagation are much bigger because of the multiple possible strength reductions). It took two weeks to complete this experiment. The proof of semantics preservation uses the same invariant as for lazy code motion and the proof remains unchanged apart from unification of operations and loads. Using the same

invariant, although effective, is questionable: it is also possible to use a simpler invariant crafted especially for constant propagation with strength reduction.

One interesting possibility is to try to abstract the invariant in the development. Instead of posing a particular invariant and then develop the framework upon it, with maybe other transformations that will luckily fit the invariant, the framework is developed with an unknown invariant on which we suppose some properties. (See Zuck [ZPL01] for more explanations.) We may hope that the resulting tool/theory be general enough for a wider class of transformations, with the possibility that the analyses have to be adapted. For example, by replacing the available expression analysis by the global value numbering algorithm of Gulwani and Necula [GN04], it is possible that the resulting validator would apply to a large class of redundancy elimination transformations.

5.4.5 Conclusion

We presented a validation algorithm for Lazy Code Motion and its mechanized proof of correctness. The validation algorithm is significantly simpler than LCM itself: the latter uses four dataflow analyses, while our validator uses only one (a standard available expression analysis) complemented with an anticipability check (a simple traversal of the CFG). This relative simplicity of the algorithm, in turn, results in a mechanized proof of correctness that remains manageable after careful proof engineering. Therefore, this work gives a good example of the benefits of the verified validator approach compared with compiler verification.

We have also shown preliminary evidence that the verified validator can be re-used for other optimizations: not only other forms of redundancy elimination, but also unrelated optimizations such as constant propagation and instruction strength reduction. More work is needed to address the validation of advanced global optimizations such as global value numbering, but the decomposition of our validator and its proof into a generic framework and an LCM-specific part looks like a first step in this direction.

> There is one last technique in the arsenal of the software optimizer that may be used to make most machines run at tip top speed. It can also lead to severe code bloat and may make for almost unreadable code, so should be considered the last refuge of the truly desperate. However, its performance characteristics are in many cases unmatched by any other approach, so we cover it here. It is called software pipelining [...]
>
> Apple Developer Connection

Chapter 6

Software pipelining

In this chapter we present a validator for a loop transformation called software pipelining. This validator is based on symbolic evaluation. Although software pipelining is a delicate transformation that heavily modifies the structure and content of loops, the power of symbolic evaluation leads to a relatively simple design and correctness proof for a validator. To this end, we reduce the problem of semantic preservation to a problem of equivalence of symbolic states and show how we can reason on symbolic states to design the validator.

6.1 Software pipelining

Software pipelining [Lam88] is an instruction scheduling optimization that exploits the instruction level parallelism in loops by overlapping successive iterations of the loop and executing them in parallel. It takes place at the RTL level. It can be performed by hand or can be implemented as a compiler pass. In the remainder of this chapter we will suppose that it is implemented as a compiler pass but results apply for both possibilities. From the bird's eye, software pipelining is performed in three steps.

Step 1 First, select the innermost loops we would like to pipeline. In our case, we focus on loops of type

$$\text{and} \quad I = 0;\ \texttt{while}\ (I < N)\ \{\ \mathcal{B}; I{+}{+}\}$$

where N is a loop invariant expression, I is incremented only once per iteration, and $\mathcal{B}$ is a sequence of non-branching instructions (meaning no conditions, no calls, no returns). Moreover the loop must not diverge. Those restrictions are standard and not specific to our experiment: software pipelining is very sensitive to the shape of loops and conditions, and can only be applied in some precise cases. We restrict the study to the case where the condition is of the form $I < N$ (unsigned).

Step 2 Next, the *software pipeliner* is called. The software pipeliner we consider here is a composition of *modulo scheduling* [Rau96, Huf93, LGAV96, RST92, MLG02] followed by *modulo variable expansion* (MVE) [Lam88]. It is implemented in several production compilers [HZ04]. It takes as its input an innermost loop that we model as

$$\begin{aligned}\texttt{type input_loop} \quad = \quad & \{\ \texttt{cond} : \texttt{instruction};\\ & \mathcal{B} : \texttt{list instruction}\ \}\end{aligned}$$

Here, `cond` is the condition of the loop and $\mathcal{B}$ its body. The software pipeliner returns a pipelined loop that we model as

$$\begin{aligned}\texttt{type output_loop} \quad = \quad & \{\ \texttt{cond} : \texttt{instruction};\\ & \mathcal{P} : \texttt{list instruction};\\ & \mathcal{B}_t : \texttt{list instruction};\\ & \mathcal{E} : \texttt{list instruction};\\ & \mu : \texttt{int};\\ & \delta : \texttt{int}\ \}\end{aligned}$$

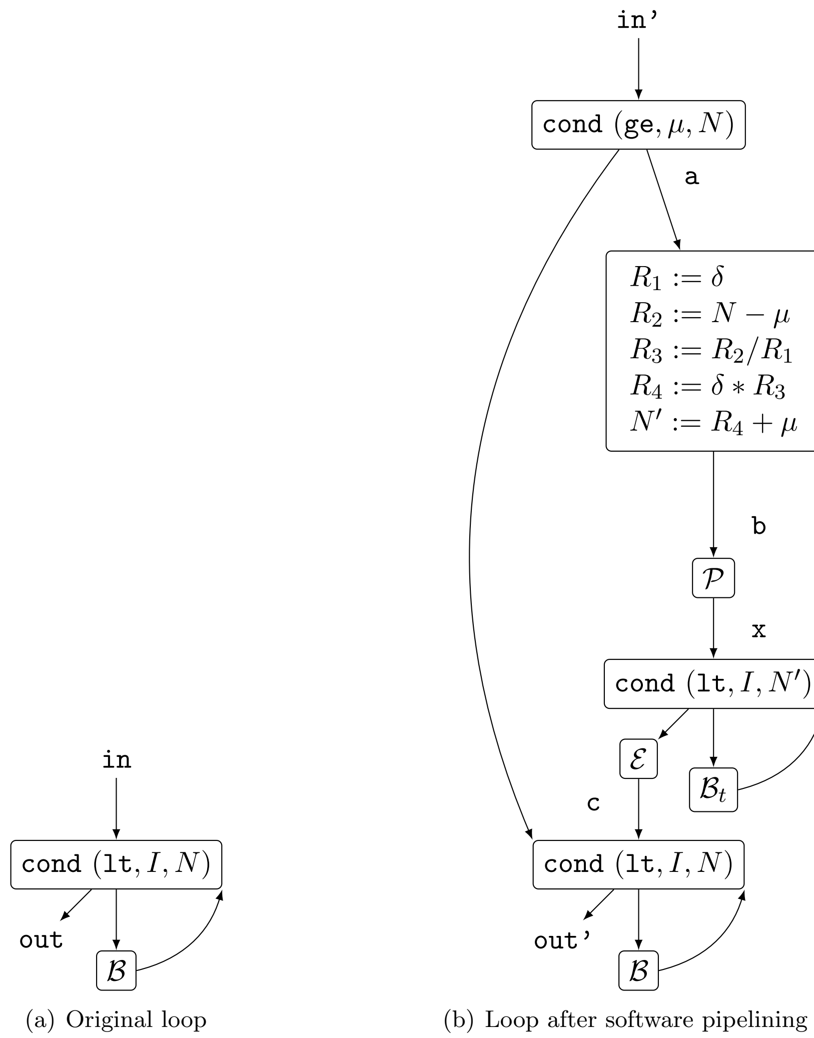
Here, $\mathcal{B}_t$ is the new loop body, also called the *steady state*. $\mathcal{P}$ is the loop prolog: a sequence of register moves (due to the modulo variable expansion) followed by a sequence of instructions that fills the pipeline until it reaches the steady state. $\mathcal{E}$ is the loop epilog: a sequence of instructions that drains the pipeline, followed by a sequence of moves. μ is the minimum number of iterations that must be performed to be able to use the pipelined loop, and δ is the amount of unrolling that has been performed on the steady state. Hence, the software pipeliner $\mathcal{SP}$ is modeled as

$$\mathcal{SP} : \texttt{input_loop} \rightarrow \texttt{output_loop}$$

We give more details of the effects of the software pipeliner on the code in section 6.1.1.

Step 3 Finally, the control-flow graph is patched to include the new pipelined loop. The global effect of software pipelining can be seen on figure 6.1.

In part (a), we sketch the loop before transformation. It has its condition at the loop entrance with its bound in register N, and the loop index in register I, and a loop body $\mathcal{B}$. In part (b), we present the control-flow graph modified to use the pipelined loop. First, a condition checks that the number of iteration is large enough by comparing μ and N. If it isn't, control flows to a copy of the original loop. If it is, a computation first sets up the bound for the pipelined loop (point `a` to point `b`). Then, control flows into the pipelined loop (point `b` to point `c`) and finally into to the copy of the original loop (point `c` to point `out'`).



(a) Original loop (b) Loop after software pipelining

Figure 6.1: High level oveview of a software pipelining transformation

Loop boundaries

In the original loop, the loop index is incremented one by one. In the pipelined loop, the loop index is incremented μ times in the prolog and epilog and δ times at each loop iteration. In both loops, the register N is invariant. Let n be the value in N during the execution of the original loop and assume it is greater or equal to μ (so that it is the pipelined loop that gets executed). Because of the unrolling, it is not always possible to execute those n iterations using the pipelined loop. We can rewrite n as $\mu + ((n-\mu)/\delta) \times \delta + (n-\mu) \mod \delta$ (recall that this is integer arithmetic). For an optimal use of the pipelined loop it must be repeated $(n-\mu)/\delta$ times, and $(n-\mu) \mod \delta$ iterations will remain after the execution of the pipelined loop. These leftover iterations are executed by the copy of the original loop.

Let $\kappa(n) = ((n-\mu)/\delta)$ be the function that computes the number of iterations that can be performed using the pipelined loop and $\rho(n) = (n-\mu) \mod \delta$ be the function that computes the number of iterations that will remain to do after the pipelined loop. In our implementation of software pipelining, we chose to let the pipelined loop iterate δ by δ and leave the condition at the loop body entrance intact. Thus, before we execute the pipelined loop, its bound must be set up to the value $\mu + \kappa(n) \times \delta$. This is what the code between point `a` and point `b` of figure 6.1 computes. Once the execution gets out of the pipelined loop, $\rho(n)$ iterations of the original loop remain to be performed. These remaining iterations are performed by the copy of the original loop.

Therefore, the value n' of the bound N' in the pipelined loop execution must be equal to $\mu + \kappa(n) \times \delta$. Also note that $\kappa(n)$ is equal to $\kappa(n')$.

6.1.1 Effects of software pipelining

Let us now focus on the software pipeliner (step 2). We will not explain how the pipeline is created: since we will validate it, only a high-level understanding of its effects is necessary to understand the remaining of the chapter. Figure 6.2 presents an example of a loop body that has been pipelined. We can understand the purpose of the loop by looking at the original code: it adds the constant f_1 to every elements of an array. The first instruction is a load from memory at the address contained in register r. The second performs the addition and leaves its result it in f_2. The third stores the value of f_2 into memory at address r. Finally, the last instruction decrements the address. Note that it is not possible to perform an effective list scheduling of this sequence of instructions, all pairs of consecutive instructions are in some dependency.

In the original loop, the load from memory is used in the addition that follows and thus the processor stalls until the load terminates. To counter this stall, the pipelined loop perform the load from memory necessary for iteration $i+1$ at iteration i. For instance, the value that is loaded from memory and stored in register f'_0 is used in the next loop iteration, 6 instructions later. This is also apparent in the prolog where we need to start two iterations for the steady

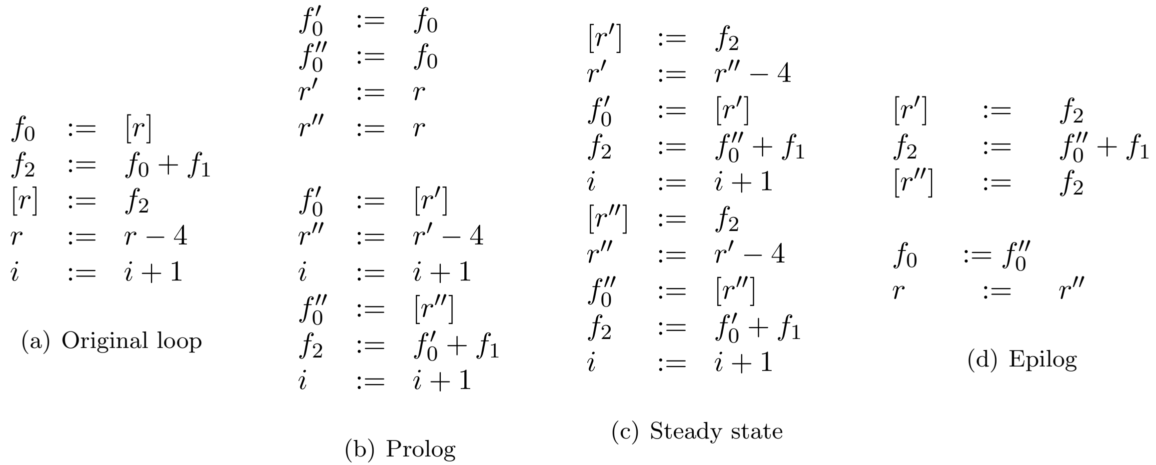


Figure 6.2: An example of pipelined loop

state to be used. A value is loaded from memory at address r_1 and before it is used, the load of address $r_1 - 4$ for the next iteration already begins.

Since a load for iteration $i + 1$ is performed on iteration i, we must carefully avoid register clashes. For instance, there must be two instances of register f_0 in the pipelined loop, f_0' and f_0''. The prolog gives an example of the register clashes that appear in the pipelined loop: there are two loads from memory that precedes the first addition. Therefore, if we use only register f_0, the second load in the prolog erases the first load, and the first addition is incorrect: it computes the value for the second iteration whereas it should compute the value of the first iteration. To avoid register clashes we perform *modulo variable expansion*. It consists in an unrolling of the loop steady state followed by a renaming of the registers that are live across several iterations. Details about modulo variable expansion can be found in [Lam88]. Because a new set of local registers are used, prolog and epilog are preceded and followed (respectively) by a sequence of moves. Note that this particular example requires some alias analysis whereas we do not treat it here: we chose this example because it is intuitive to think of an array in memory. (In fact, alias analysis is important and should be performed but this is a problem in its own right.)

To summarize, the software pipeliner performs a set of modifications which make it very difficult to relate the states of the pipelined loop with the original one: it restructures the control of the loop by adding a prolog, an epilog and unrolling the loop; it schedules the instructions such that distinct iterations of the original loop are interleaved; adds and renames registers such that some values now span several loop iterations. Consequently, it appears difficult to formalize the relation that holds between the two loops during their executions to prove semantic preservation. The bottom line of the present chapter is that *a posteriori* validation of software pipelining is feasible, and even relatively simple, by judicious use of symbolic evaluation.

6.2 Overview of the design

The problem we consider in this work is the design and implementation of a validator that enforces semantic preservation for the software pipeliner (step 2 of section 6.1). We chose not to validate the complete software pipelining transformation but only the software pipeliner for two reasons. First, from a testing and debugging viewpoint, the only part of the transformation that is delicate is the software pipeliner. The code generated to check that there are enough iterations to use the pipelined loop and the code generated to set up the bounds for the pipelined loop are easy to test and debug. Besides, a quick manual inspection can attest that the default loop in the transformed graph is a copy of the original loop. Second, from a formal verification viewpoint, the use of a verified validator seems to be a reasonable choice for the software pipeliner, but a direct proof of correctness is better suited for the modification that we make on the graph to use the pipelined loop (step 3 of section 6.1). Nevertheless, in order to state the software pipeliner specification and prove its correctness, we need to assume that the inclusion of the pipelined loop is correctly done and thus assume a few preconditions. In section 6.6, we give a comprehensive and precise list of those preconditions.

6.2.1 A sound symbolic model

A key concern is that the registers used in the original and pipelined loop are not the same: the latter uses a set of new registers introduced by MVE. The equivalence predicate between states – which are composed of the register file and the memory – of the original loop and the pipelined one will thus be defined over a set of observable registers, the ones that need to be equivalent before and after the loops are executed. The finite set θ of observable registers contains all the registers used by the original code except N, the register that carries the loop boundary (hence, θ contains all the registers of the transformed code except N, and the registers used by the MVE). Formally, the equivalence between two states, noted $S \cong_\theta T$, is defined as:

$$\frac{\forall R \in \theta, S.r(R) = T.r(R) \qquad S.m = T.m}{S \cong_\theta T}$$

where $S.r$ denotes the register file and $S.m$ denotes the memory.

Our goal is to prove that semantics are preserved. Consider again figure 6.1 and assume that the original loop starts at point `in` in state S, the pipelined loop at point `in'` in state T and such that both states are equivalent $S \cong_\theta T$. If the original loop performs an execution from S to some state S' at point `out`, then there must exists a state T' at point `out'` such that the pipelined loop execution goes from T to T' and such that $S' \cong_\theta T'$.

The key ingredient in our design is symbolic evaluation. Again, symbolic evaluation is a function (written α) that takes a list of non-branching instructions as its input and returns

the code using another representation. The interest of symbolic evaluation is that this new code representation, which we call a symbolic state, is less sensitive to syntactic details – in particular, all the equivalent scheduling of some sequence of instructions l will all have the same representation $\alpha(l)$. This is why we believe symbolic evaluation is a tool of choice to validate a transformation such as software pipelining: we can verify semantic preservation for two sequences of instructions by symbolically evaluating them and check whether their corresponding symbolic states are equivalent. As semantic preservation is defined on the set of observables θ, so is equivalence between symbolic states. This equivalence, written $\alpha(l_1) \approx_\theta \alpha(l_2)$, states that all the registers that belong to θ hold the same symbolic values. The differences between the symbolic evaluation presented in chapters 3 and 4 and the one presented in this chapter result from this generalization. In particular, the theorem that relate the concrete execution of a list of non-branching instructions and its symbolic evaluation needs to be generalized and requires new operators to be defined on symbolic states. Symbolic evaluation and its extension to handle observables is presented in details in section 6.3.

For a given number of iterations n, executing the original loop is the same as executing the sequence of code composed of $\mu+\kappa(n)\times\delta+\rho(n)$ unrolling of the initial loop body, which we write as $\mathcal{B}^{\mu+\kappa(n)\times\delta+\rho(n)}$. Likewise, executing the pipelined loop is the same as executing the sequence of code $\mathcal{P}\mathcal{B}_t^{\kappa(n)}\mathcal{E}\mathcal{B}^{\rho(n)}$ where $\mathcal{B}_t$ has been unrolled $\kappa(n)$ times, followed by $\rho(n)$ iterations of $\mathcal{B}$. Those two sequences of instructions must have the same semantics, and the latter is just a scheduling of the former; thus, we must have $\alpha(\mathcal{B}^{\mu+\kappa(n)\times\delta+\rho(n)}) \approx_\theta \alpha(\mathcal{P}\mathcal{B}_t^{\kappa(n)}\mathcal{E}\mathcal{B}^{\rho(n)})$. The last $\rho(n)$ iterations of $\mathcal{B}$ are obviously equivalent so we only need to pay attention to the first $\mu+\kappa(n)\times\delta$ iterations. Generalizing to any number of iterations, we obtain a simple characterization of how the two loops must be related so that semantics is preserved:

$$H_1 : \quad \forall n, \alpha(\mathcal{B}^{\mu+\kappa(n)\times\delta}) \approx_\theta \alpha(\mathcal{P}\mathcal{B}_t^{\kappa(n)}\mathcal{E})$$

This equivalence is useful as long as the loop indexes behave so that if the original loop is executed $\mu + \kappa(n) \times \delta$ the pipelined loop is really executed $\kappa(n)$ times. Thankfully, we can state, through symbolic evaluation, how the loop indexes need to be related so that they are "synchronous".

$$H_2 : \quad \forall n, \alpha(\mathcal{B}^{\mu+\kappa(n)\times\delta})!I = \alpha(\mathcal{P}\mathcal{B}_t^{\kappa(n)})!I$$

where $\alpha(l)!I$ denotes the symbolic value in register I of the symbolic state $\alpha(l)$.

These first properties tells us that both loops perform the same state transformation on the subset of registers that belongs to θ. They must also start with the same input and the start states are only equivalent on the registers θ themselves, not necessarily on the others. Thus, we must make sure that the symbolic states $\alpha(\mathcal{B}^{\mu+\kappa(n)\times\delta})$ and $\alpha(\mathcal{P}\mathcal{B}_t^{\kappa(n)}\mathcal{E})$ are closed under θ: all the symbolic values in registers that belong to θ must be independent of the registers that do not belong to θ. We write this property $\langle\alpha(l) \mid \theta\rangle$, meaning that the symbolic state $\alpha(l)$ is

closed under θ. In fact, from a soundness point of view, we only need this property to hold for the pipelined loop, that is:

$$H_3 : \quad \forall n, \langle \alpha(\mathcal{P}\mathcal{B}_t^{\kappa(n)}\mathcal{E}) \mid \theta \rangle$$

H_3 states that the registers used to avoid register clashes (the new registers that did not exist in the original code) do not escape: the state reached by the pipelined loop does not depend on the value of those registers before the pipelined loop execution. Those registers are only used within the pipelined loop.

A crucial observation is that if the pipelined loop is correctly used in the transformed program, then the satisfiability of these three symbolic formulas enforce semantic preservation between the original and pipelined loops. We detail in section 6.6 what are the preconditions that a correct use of the pipelined loop must set up and prove that the satisfiability of the symbolic model of the relation between the loops implies semantic preservation between the loops.

6.2.2 Satisfiability of the model

To check whether two sequences of code have the same symbolic evaluation, we have at our disposal a function `eq` that takes as input two symbolic states along with a set of observables and returns a Boolean such that $\texttt{eq}(\alpha(l_1), \alpha(l_2), \theta) = \mathit{true}$ implies $\alpha(l_1) \approx_\theta \alpha(l_2)$. Unfortunately, we cannot use it to verify H_1 because of the universal quantification over the number n of iterations, nor can we verify H_2 or H_3.

The key idea to be able to check the validity of the symbolic formulas is to express a symbolic relation between the two loops. It turns out that, because symbolic evaluation hides syntactic details, it is possible to express simply a high-level relation between a loop and its pipelined version. For H_1, this property is based on the idea that if a software pipelining is correct, then it must always be possible, when enough iterations remain to be performed, to choose between:

1. Go one more time through the pipelined loop body $\mathcal{B}_t$ and then leave the pipelined loop by going through $\mathcal{E}$;
2. Leave the pipelined loop by going through $\mathcal{E}$ and then perform δ iterations of $\mathcal{B}$.

More formally, $\alpha(\mathcal{E}\mathcal{B}^\delta) \approx_\theta \alpha(\mathcal{B}_t\mathcal{E})$ should hold. Also, when there are only μ iterations, it is necessary that going μ times though $\mathcal{B}$ is the same as going through $\mathcal{P}\mathcal{E}$. More formally, $\alpha(\mathcal{B}^\mu) \approx_\theta \alpha(\mathcal{P}\mathcal{E})$. To summarize, the original and pipelined loop should satisfy the following properties:

$$\alpha(\mathcal{E}\mathcal{B}^\delta) \approx_\theta \alpha(\mathcal{B}_t\mathcal{E}) \text{ and } \alpha(\mathcal{B}^\mu) \approx_\theta \alpha(\mathcal{P}\mathcal{E})$$

These properties imply H_1 and can be checked using `eq`. This is the cornerstone of the validator.

To synchronize the counters, we rely on the fact that I should be incremented δ times in $\mathcal{B}_t$ and μ times in the prolog. (We consider that the μ increments happen within the prolog, as is done by our software pipeliner implementation. To be more general, we should verify that the μ increments are split between the prolog and epilog.) Thus, the counters should satisfy the following properties:

$$\alpha(\mathcal{B}^{\mu})!I = \alpha(\mathcal{P})!I \text{ and } \alpha(\mathcal{B}^{\delta})!I = \alpha(\mathcal{B}_t)!I$$

Again, those formulas are a sound characterization of H_2 and we can check their validity.

Finally, H_3 is a consequence of H_1 and of the fact that θ must be chosen so that $\langle \mathcal{B} \mid \theta \rangle$. Our validator must check this condition using the Boolean checker `dom` which, given a symbolic state, verifies that it is closed under θ.

The design of the validator results from those properties: it checks that all the following verifications return `true`:

- `eq`$(\alpha(\mathcal{B}_t\mathcal{E}), \alpha(\mathcal{E}\mathcal{B}^{\delta}), \theta)$
- `eq`$(\alpha(\mathcal{B}^{\mu}), \alpha(\mathcal{P}\mathcal{E}), \theta)$
- $\alpha(\mathcal{B}^{\mu})!I = \alpha(\mathcal{P})!I$,
- $\alpha(\mathcal{B}^{\delta})!I = \alpha(\mathcal{B}_t)!I$
- `dom`$(\alpha(\mathcal{B}), \theta)$

This algorithm is presented with more details in section 6.5.2 together with a proof of soundness, a comment on completeness, the resulting validator and the resulting proof of correctness of the validator. To make the proof as simple as possible, we rely on the fact that, in some sense, symbolic state are state transformers that can be composed. But since all the comparisons are up to a set of observables, we must pay attention to the conditions under which we can rewrite symbolic states. This is studied in details in section 6.4.

6.3 Symbolic evaluation modulo observables

In this section, we recall the basic principles and properties of symbolic evaluation with a slight generalization to take into account comparison of symbolic states modulo a set of observables θ. This requires to introduce a new predicate over symbolic states that checks whether a state is closed under θ. A symbolic state is closed under θ if for every resource (register or memory) of the state that belongs to θ, the symbolic value corresponding to that resource has all its variables included in θ. Equipped with this predicate, we can state the soundness theorem of symbolic evaluation generalized for equivalence modulo observables. All the operations and basic properties are presented in figure 6.3.

α	:	`seq` $\rightarrow$ `sst`
ε	:	`sst`
`!`	:	`sst` $\rightarrow$ `reg` $\rightarrow$ `elt`
`eq`	:	`observables` $\rightarrow$ `sst` $\rightarrow$ `sst` $\rightarrow$ `bool`
`dom`	:	`observables` $\rightarrow$ `sst` $\rightarrow$ `bool`
$\sim$	:	`sst` $\rightarrow$ `sst` $\rightarrow$ `Prop`
$\approx$	:	`observables` $\rightarrow$ `sst` $\rightarrow$ `sst` $\rightarrow$ `Prop`
$\langle \mid \rangle$	:	`sst` $\rightarrow$ `observables` $\rightarrow$ `Prop`

(a) Operators

(1)	$\forall x,$	$x \sim x$
(2)	$\forall xy,$	$x \sim y \Rightarrow y \sim x$
(3)	$\forall xyz,$	$x \sim y \Rightarrow y \sim z \Rightarrow x \sim z$
(4)	$\forall x,$	$x \approx_\theta x$
(5)	$\forall xy,$	$x \approx_\theta y \Rightarrow y \approx_\theta x$
(6)	$\forall xyz,$	$x \approx_\theta y \Rightarrow y \approx_\theta z \Rightarrow x \approx_\theta z$
(7)	$\forall xyz,$	$x \sim y \Rightarrow x \approx_\theta z \Rightarrow y \approx_\theta z$
(8)	$\forall xyz,$	$y \sim z \Rightarrow x \approx_\theta y \Rightarrow x \approx_\theta z$
(9)	$\forall xy,$	$x \approx_\theta y \Rightarrow \langle x \mid \theta \rangle \Rightarrow \langle y \mid \theta \rangle$
(10)	$\forall xyr,$	$x \sim y \Rightarrow x!r = y!r$
(11)	$\forall xy,$	$\texttt{eq}(\theta, x, y) = \texttt{true} \Rightarrow x \approx_\theta y$
(12)	$\forall x,$	$\texttt{dom}(\theta, x) = \texttt{true} \Rightarrow \langle x \mid \theta \rangle$

(b) Properties

Figure 6.3: Operators and basic properties of symbolic evaluation

Let `sst` be the type of symbolic states. A symbolic state is composed of three elements: a map from registers to symbolic values, a symbolic memory, and a set of constraints. It is almost the same as presented in chapter 3 but for the RTL intermediate language.

Symbolic value expressions:

$t ::= R^0$ initial value of register R
$\quad \mid \texttt{Op}(op, \vec{t})$
$\quad \mid \texttt{Load}(\kappa, mode, \vec{t}, t_m)$

Symbolic memory expressions:

$t_m ::= \texttt{Mem}^0$ initial memory store
$\quad \mid \texttt{Store}(\kappa, mode, \vec{t}, t_m, t)$

Symbolic register file:

$r ::= R \mapsto t$

Symbolic memory:

$m ::= t_m$

Constraints:

$s ::= \{t, t_m, \ldots\}$

The symbolic evaluation function, α, is a function that takes an element of type `seq` (a list of non-branching instructions) and returns a symbolic state. Its definition is similar to the

symbolic evaluation of chapter 3 except that the move operations are executed.

$$\alpha(i_1 i_2 ... i_n) = \alpha_x(i_n, \ldots \alpha_x(i_2, \alpha_x(i_1, \varepsilon)) \ldots)$$

with ε being the initial state and

$$\begin{array}{l}
\texttt{updateFile}(R, t, (r, m, s)) \\
\quad = \quad (r\{R \leftarrow t\},\ m,\ s \cup \{t\}) \\
\texttt{updateMem}(t_m, (r, m, s)) \\
\quad = \quad (r,\ t_m,\ s \cup \{t_m\}) \\
\alpha_x(\texttt{op}(\texttt{move}, R_s, R), (r, m, s)) \\
\quad = \quad (r\{R \leftarrow r(R_s)\}, m, s) \\
\alpha_x(\texttt{op}(op, \vec{R}, R), (r, m, s)) \\
\quad = \quad \texttt{updateFile}(R, \texttt{Op}(op, r(\vec{R})), (r, m, s)) \\
\alpha_x(\texttt{load}(\kappa, mode, \vec{R}, R), (r, m, s)) \\
\quad = \quad \texttt{updateFile}(R, \texttt{Load}(\kappa, mode, r(\vec{R}), m, (r, m, s)) \\
\alpha_x(\texttt{store}(\kappa, mode, \vec{R}, R), (r, m, s)) \\
\quad = \quad \texttt{updateMem}(\texttt{Store}(\kappa, mode, r(\vec{R}), m, r(R)), (r, m, s))
\end{array}$$

We use two notions of equivalence between symbolic states. The first one, written $\sim$, is defined as:

$$(r, m, s) \sim (r', m', s')$$

if and only if

$$\forall R, r(R) = r'(R) \text{ and } m = m' \text{ and } s = s'$$

It is used to reason on the structure of one symbolic states. It is reflexive, symmetric and transitive (properties 1 to 3 in figure 6.3). The other equivalence, written $\approx_\theta$, is parametrized by the set of observables θ and defined as

$$(r, m, s) \approx_\theta (r', m', s')$$

if and only if

$$\forall R \in \theta, r(R) = r'(R) \text{ and } m = m' \text{ and } s = s'$$

It is used to compare two distinct symbolic states up to the observable registers θ. It is reflexive,

symmetric and transitive (properties 4 to 6 in figure 6.3). The relation $\sim$ is compatible with the relation $\approx_\theta$ (properties 7 and 8 in figure 6.3).

We need to be able to express the fact that a set of symbolic values has its variables included in the observables. Let $Res(t)$ be the set of resources that appear within t. The two following rules give the flavor of Res's definition:

$$Res(R^0) = \{R\} \qquad\qquad Res(\texttt{Op}(op, t_1...t_n)) = Res(t_1) \cup ... \cup Res(t_n)$$

From this predicate, we can define the predicate $\langle x \mid \theta \rangle$ which states that all the symbolic terms carried by registers in θ of a state x have all their symbolic variables included in θ. As an example, for every symbolic tree $r(\rho)$ carried by a register $\rho \in \theta$ that we want to observe, all the registers that appear in $r(\rho)$ must themselves belong to the set of observables.

$$\langle (r, m, s) \mid \theta \rangle$$

if and only if

$$\forall R \in \theta,\ Res(r(R)) \subseteq \theta \text{ and } Res(m) \subseteq \theta \text{ and } \forall t \in s, Res(t) \subseteq \theta$$

The relation $\approx_\theta$ is compatible with $\langle | \rangle$ (property 9 in figure 6.3).

The Boolean functions `eq` and `dom` are checkers for predicates $\approx$ and $\langle | \rangle$ (properties 11 and 12 in figure 6.3). They are simple to define because θ is a finite set.

The soundness theorem says that, starting from equivalent states (in the concrete semantics), two sequences of non-branching instructions l_1 and l_2 that are equivalent up to symbolic evaluation lead to equivalent states (again, in the concrete semantics). The concrete execution of a sequence of non-branching instructions l from state S to S' is noted $l : S \xrightarrow{*} S'$. It is the reflexive and transitive closure of the semantics of the subset of the instructions that are non-branching. Because of the generalization of this theorem to observables, we must add the hypothesis that the symbolic evaluation of l_2 is closed under the observables θ.

Theorem 6.1. *Let l_1 and l_2 be two lists of non-branching instructions. Assume $\alpha(l_1) \approx_\theta \alpha(l_2)$, $\langle \alpha(l_2) \mid \theta \rangle$, $S \cong_\theta T$ and l_1: $S \xrightarrow{*} S'$. Then, there exists a state T' such that l_2: $T \xrightarrow{*} T'$ and $S' \cong_\theta T'$.*

Proof. Using theorems similar to theorem 3.1 and 3.2 in chapter 3 we can deduce from $\alpha(l_1) \approx_\theta \alpha(l_2)$ and l_1: $S \xrightarrow{*} S'$ that there exists a state S'' such that l_2: $S \xrightarrow{*} S''$ and $S \cong_\theta S''$. Then, we prove that since $S \cong_\theta T$ and $\langle \alpha(l_2) \mid \theta \rangle$ there exists a state T' such that l_2: $T \xrightarrow{*} T'$ and $S'' \cong_\theta T'$. With $S \cong_\theta S''$ and $S'' \cong_\theta T'$, the result follows. □

In order to reason about control, we need a way to observe the symbolic value that a particular register holds in some symbolic state. This is achieved by the "get" operator, written !,

and defined as

$$(r, m, c)!R = r(R)$$

The Boolean comparison of two symbolic values and their equivalence predicate are just syntactic equality. The get function has the property that it is compatible with the equivalence (property 10 in figure 6.3).

The soundness theorem states that if two symbolic states assign the same symbolic value to a register, then the corresponding executions will lead to the same concrete value in this register.

Theorem 6.2. *Let l_1 and l_2 be two sequences of non-branching instructions. Assume $\alpha(l_1)!R = \alpha(l_2)!R$, $S \cong_\theta T$, $l_1{:}S \xrightarrow{*} S'$ and $l_2{:}T \xrightarrow{*} T'$. Then, $S'.r(R) = T'.r(R)$.*

6.4 Reasoning over symbolic evaluations

In this section, we define new operators over symbolic evaluations and prove key properties that enable to abstract over symbolic evaluation representation and make the forthcoming proofs easier. We first define a composition operator $\circ$ over symbolic states such that, given two sequences of instructions l_1 and l_2, we have $\alpha(l_1 l_2) \sim \alpha(l_1) \circ \alpha(l_2)$. Then we state two properties which show that it is possible to rewrite on the right – if $y \approx_\theta z$ then $x \circ y \approx_\theta x \circ z$ – and on the left, if the right operand of the composition is closed under θ – if $x \approx_\theta y$ and $\langle z \mid \theta \rangle$ then $x \circ z \approx_\theta y \circ z$–.

6.4.1 Decomposition

Let t be a symbolic term and (r, m, s) be a symbolic state. The substitution $t[(r, m, s)]$ of resources of t by (r, m, s) is defined as:

$$
\begin{aligned}
& R^0[(r,m,s)] = \\
& \qquad r(R^0) \\
& \texttt{Op}(op, \vec{t})[(r,m,s)] = \\
& \qquad \texttt{Op}(op, \texttt{map}(\lambda e \Rightarrow e[(r,m,s)])\ \vec{t}) \\
& \texttt{Load}(\kappa, mode, \vec{t}, t_m)[(r,m,s)] = \\
& \qquad \texttt{Load}(\kappa, mode, \texttt{map}(\lambda e \Rightarrow e[(r,m,s)])\ \vec{t}, t_m[(r,m,s)]) \\
& \texttt{Mem}^0[(r,m,s)] = \\
& \qquad m \\
& \texttt{Store}(\kappa, mode, \vec{t}, t_m, t)[(r,m,s)] = \\
& \qquad \texttt{Store}(\kappa, mode, \texttt{map}(\lambda e \Rightarrow e[(r,m,s)])\ \vec{t}, t_m[(r,m,s)], t[(r,m,s)])
\end{aligned}
$$

The composition operator, $\circ$, is defined as:

$$
\begin{array}{lll}
(1) & \forall xyz, & x \circ (y \circ z) \sim (x \circ y) \circ z \\
(2) & \forall xyz, & x \sim y \Rightarrow x \circ z \sim y \circ z \\
(3) & \forall xyz, & y \sim z \Rightarrow x \circ y \sim x \circ z \\
\\
(4) & \forall x, & x \circ \varepsilon \sim x \\
(5) & \forall x, & \varepsilon \circ x \sim x \\
\\
(6) & & \alpha([]) \sim \varepsilon \\
(7) & \forall l_1 l_2 & \alpha(l_1 l_2) \sim \alpha(l_1) \circ \alpha(l_2) \\
\\
(8) & \forall xyz, & y \approx_\theta z \Rightarrow x \circ y \approx_\theta x \circ z \\
(9) & \forall xyz, & \langle z \mid \theta \rangle \Rightarrow x \approx_\theta y \Rightarrow x \circ z \approx_\theta y \circ z \\
\\
(10) & \forall xy, & \langle x \mid \theta \rangle \Rightarrow \langle y \mid \theta \rangle \Rightarrow \langle x \circ y \mid \theta \rangle \\
(11) & \forall xyr, & Res(y!r) \subseteq r \Rightarrow (x \circ y)!r = y!r[x]
\end{array}
$$

Figure 6.4: The composition operator and its properties

$$
\begin{array}{l}
(r_1, m_1, s_1) \circ (r_2, m_2, s_2) = \\
\qquad (R \mapsto r_2(R)[(r_1, m_1, s_1)] \ , \ m_2[(r_1, m_1, s_1)] \ , \ s_1 \cup \{e[(r_1, m_1, s_1)] \mid e \in s_2\})
\end{array}
$$

The $\circ$ operator is built upon the substitution $t[x]$ (x is a symbolic state and t a symbolic value). As a consequence, proving properties about $\circ$ comes down to proving the restriction of the property to $t[x]$. For example, one important theorem that follows is the associativity of $\circ$ (property 1 in figure 6.4). The key step is to prove the associativity for a single term: $t[x \circ y] = (t[y])[x]$. Once this is proved, generalizing the proof to the definition of $\circ$ is obvious, tedious and finally of no interest to the understanding. (Tedious because an induction is to be performed on the structure of the map and on the sets.) In the remainder we will focus on the key proofs: the ones for single terms. Note that we are doing an abuse of notation in the definition of $t[x]$. The substituted terms are of different types so there should be several versions of $t[x]$: one for symbolic values and one for symbolic memories. Thus, it should be noted that proofs by induction on the symbolic term t are proof by mutual induction over symbolic terms and symbolic memories. In the remainder we avoid this detail for simplicity.

The composition operator $\circ$ is not only associative, but also compatible with the relation $\sim$ (properties 2 and 3 in figure 6.4). Hence, we have the property that $\forall x, y, u, v, x \sim y \wedge u \sim v \Rightarrow x \circ u \sim y \circ v$. Moreover, the initial state ε is a neutral element for $\circ$ (properties 4 and 5 in figure 6.4).

To summarize, the set of symbolic states together with the operator $\circ$, the neutral element

ε and the relation $\sim$ form a quotient monoid. Moreover, since the set of lists of non-branching instructions together with the empty list and the list concatenation form a monoid, we prove that α is a monoid morphism from non-branching instruction lists to symbolic states (properties 6 and 7 in figure 6.4). Property 7, which we call the *decomposition* property, implies that the evaluation of a sequence of instructions can be decomposed into the composition of the evaluations of its constituents.

To prove this lemma, first note that, by induction on l_1,

$$\alpha_x(l_1 l_2, t) \sim \alpha_x(l_2, \alpha_x(l_1, t))$$

Moreover, by induction on the symbolic term t, we prove that abstracting a single instruction j from a symbolic state x is equivalent to composing the state x with the abstraction of j:

$$\alpha_x(j :: \texttt{nil}, x) \sim x \circ (\alpha_x(j :: \texttt{nil}, \varepsilon))$$

We can deduce, by induction on l, the property that $\circ$ is distributive over α_x, that is:

$$\alpha_x(l, x \circ y) \sim x \circ \alpha_x(l, y)$$

In conclusion, we have

$$\begin{array}{rcll} \alpha(l_1 l_2) & \sim & \alpha_x(l_1 l_2, \varepsilon) & \text{By definition} \\ & \sim & \alpha_x(l_2, \alpha_x(l_1, \varepsilon)) & \\ & \sim & \alpha_x(l_2, \alpha_x(l_1, \varepsilon) \circ \varepsilon) & \text{By lemma 15} \\ & \sim & \alpha_x(l_1, \varepsilon) \circ \alpha_x(l_2, \varepsilon) & \text{By distributivity} \\ & \sim & \alpha(l_1) \circ \alpha(l_2) & \text{By definition} \end{array}$$

This decomposition of symbolic evaluation is one key to ease reasoning on symbolic evaluation, but we also need to know how we can rewrite symbolic states in a symbolic formula.

6.4.2 Rewriting of symbolic states

It is obvious that we can rewrite on the right (property 8 in figure 6.4), that is

$$y \approx_\theta z \Rightarrow x \circ y \approx_\theta x \circ z$$

Consider a register R such that $R \in \theta$. Since $(r_y, m_y, c_y) \approx_\theta (r_z, m_z, c_z)$ we have $r_y(R) = r_z(R)$. In $(r, m, s) \circ (r_y, m_y, s_y)$, R is associated with $r_y(R)[(r, m, s)]$ which is equal to $r_z(R)[(r, m, s)]$.

However, we cannot prove the same kind of property to rewrite on the left of $\circ$. Suppose that t has a leaf with a symbolic register R that does not belong to θ. Then, the left substitution replaces this symbolic register by $r_1(R)$ and the right substitution by $r_2(R)$. Since R does not

belong to θ, it is not true that $r_1(R) = r_2(R)$ and the result of the substitution may differ. A solution is to suppose not only that $(r_1, m_1, s_1) \approx_\theta (r_2, m_2, s_2)$ but also that t has all its variables in θ, that is, $Res(t) \subseteq \theta$. That way, all the symbolic resources that are substituted are identical in the substituting symbolic state. From those observation, property 9 of figure 6.4 follows:

$$\langle z \mid \theta \rangle \Rightarrow x \approx_\theta y \Rightarrow x \circ z \approx_\theta y \circ z$$

Finally, given two symbolic states x and y and a register R such that the only variable that appears in the symbolic value $y!R$ is R^0, $(Res(y!R) \subseteq \{R\})$, then, the symbolic value $(x \circ y)!R$ is equal to the substitution of R^0 by $x!R$ in $y!r$ (property 11 in figure 6.4).

6.5 A validator for the software pipeliner

In this section, we present the validator V that takes an original loop i, a pipelined loop o and the set θ as inputs and returns a Boolean along with a proof that it enforces properties H_1, H_2 and H_3.

The formulas H_1, H_2 and H_3 are universally quantified and as such we can not symbolically evaluate them, at least not with the operators at our disposal. However, we have presented a few symbolic properties that relate the original loop and its pipelined version, that can be checked, and that are provably sound, as shown in this section.

In section 6.5.2 we recall the properties that can be checked for and enforce H_1 and H_2. Then, in section 6.5.2, we present the resulting validator and its proof.

6.5.1 Finite characterizations

The first property states that for any number of iterations, the codes $\mathcal{B}^{\mu+k\times\delta}$ and $\mathcal{P}\mathcal{B}_t^k\mathcal{E}$ are symbolically equivalent if and only if the code that computes $\mathcal{B}$ μ times is equivalent to the code that computes the prolog and the epilog without going into the pipelined loop body; the code that computes $\mathcal{B}_t$ and then get out of the pipelined loop (epilog plus registers moves) is symbolically equivalent to the code that gets out of the pipelined loop and then computes δ $\mathcal{B}$.

$$\forall n, \alpha(\mathcal{B}^{\mu+n\times\delta}) \approx_\theta \alpha(\mathcal{P}\mathcal{B}_t^n\mathcal{E})$$

if and only if

$$\alpha(\mathcal{B}^{\mu}) \approx_\theta \alpha(\mathcal{P}\mathcal{E}) \text{ and } \alpha(\mathcal{B}_t\mathcal{E}) \approx_\theta \alpha(\mathcal{E}\mathcal{B}^{\delta})$$

The "if" part (soundness) of this claim is easy to show. The only point that we must pay attention to is whether symbolic states are closed under θ when rewriting on the left of a composition.

Theorem 6.3. *If* $\alpha(\mathcal{B}^{\mu}) \approx_{\theta} \alpha(\mathcal{P}\mathcal{E})$ *and* $\alpha(\mathcal{B}_t\mathcal{E}) \approx_{\theta} \alpha(\mathcal{E}\mathcal{B}^{\delta})$*, then, for all* n, $\alpha(\mathcal{B}^{\mu+n\times\delta}) \approx_{\theta} \alpha(\mathcal{P}\mathcal{B}_t^n\mathcal{E})$.

Proof. By induction on n.
If $n = 0$, trivial.
We now suppose IH: $\alpha(\mathcal{B}^{\mu+n\times\delta}) \approx_{\theta} \alpha(\mathcal{P}\mathcal{B}_t^n\mathcal{E})$,
and prove that $\alpha(\mathcal{B}^{\mu+(n+1)\times\delta}) \approx_{\theta} \alpha(\mathcal{P}\mathcal{B}_t^{n+1}\mathcal{E})$

$$\begin{array}{rcll}
\alpha(\mathcal{B}^{\mu+(n+1)\times\delta}) & \approx_{\theta} & \alpha(\mathcal{B}^{\mu+n\times\delta}) \circ \alpha(\mathcal{B}^{\delta}) & \text{by decomposition}\\
& \approx_{\theta} & \alpha(\mathcal{P}\mathcal{B}_t^n\mathcal{E}) \circ \alpha(\mathcal{B}^{\delta}) & \text{by left rewriting of the IH}\\
& \approx_{\theta} & \alpha(\mathcal{P}\mathcal{B}_t^n) \circ \alpha(\mathcal{E}\mathcal{B}^{\delta}) & \text{by recomposition/decomposition}\\
& \approx_{\theta} & \alpha(\mathcal{P}\mathcal{B}_t^n) \circ \alpha(\mathcal{B}_t\mathcal{E}) & \text{by right rewriting}\\
& \approx_{\theta} & \alpha(\mathcal{P}\mathcal{B}_t^{n+1}\mathcal{E}) & \text{by recomposition}
\end{array}$$

□

We now argue informally why we believe the "only if" part (completeness) of the proof holds. Consider some number of iteration n. From H_1, we have $\alpha(\mathcal{B}^{\mu+n\times\delta}) \approx_{\theta} \alpha(\mathcal{P}\mathcal{B}_t^n\mathcal{E})$. If the loop performs one more iteration (that is $n+1$), we have $\alpha(\mathcal{B}^{\mu+(n+1)\times\delta}) \approx_{\theta} \alpha(\mathcal{P}\mathcal{B}_t^{(n+1)}\mathcal{E})$. From those two equivalences, we deduce the following equivalence:

$$\alpha(\mathcal{P}\mathcal{B}_t^n) \circ \alpha(\mathcal{E}\mathcal{B}^{\delta}) \approx_{\theta} \alpha(\mathcal{P}\mathcal{B}_t^n) \circ \alpha(\mathcal{B}_t\mathcal{E})$$

Of course, this does not imply that $\alpha(\mathcal{B}_t\mathcal{E}) \approx_{\theta} \alpha(\mathcal{E}\mathcal{B}^{\delta})$. To understand why, consider the case $\alpha(R_1 := 1; R_2 := 1) \approx_{\{R_1,R_2\}} \alpha(R_1 := 1; R_2 := R_1)$: the equivalence $\alpha(R_2 := 1) \approx_{\{R_1,R_2\}} \alpha(R_2 := R_1)$ is not true.

In our case, $\alpha(\mathcal{B}_t\mathcal{E})$ or $\alpha(\mathcal{E}\mathcal{B}^{\delta})$ could make use of some property of the state set up by $\alpha(\mathcal{P}\mathcal{B}_t^n)$. But this must work for any number of iterations, not only n and $n+1$ and so this property must be an invariant.

In conclusion, if the software pipeliner makes use of some loop invariant, then, it may not be true that $\alpha(\mathcal{B}_t\mathcal{E}) \approx_{\theta} \alpha(\mathcal{E}\mathcal{B}^{\delta})$. However, we do not consider this as a problem because usual software pipeliners are purely syntactic transformations that do not exploit loop invariants.

The second property states that, for any number of iterations k, the symbolic value assigned to I by $\alpha(B^{\mu+n\times\delta})$ and $\alpha(\mathcal{P}\mathcal{B}_t^n)$ are identical if and only if the prolog and μ repetitions of the original loop symbolically perform the same transformation on I; the original loop body repeated δ times and the transformed loop body perform the same symbolic transformation on I.

$$\forall n, \alpha(\mathcal{B}^{\mu+n\times\delta})!I = \alpha(\mathcal{P}\mathcal{B}_t^n)!I$$

if and only if

$$\alpha(\mathcal{B}^{\mu})!I = \alpha(\mathcal{P})!I \text{ and } \alpha(\mathcal{B}^{\delta})!I = \alpha(\mathcal{B}_t)!I$$

It is easy to prove the "if" part of this claim (soundness).

Theorem 6.4. *If* $\alpha(\mathcal{B}^{\mu})!I = \alpha(\mathcal{P})!I$ *and if* $\alpha(\mathcal{B}_t)!I = \alpha(\mathcal{B}^{\delta})!I$*, then, for all* n, $\alpha(\mathcal{B}^{\mu+n\times\delta})!I = \alpha(\mathcal{P}\mathcal{B}_t^n)!I$.

Proof. By induction on n.
If $n = 0$.
We now suppose IH: $\alpha(\mathcal{B}^{\mu+n\times\delta})!I = \alpha(\mathcal{P}\mathcal{B}_t^n)!I$,
and prove that $\alpha(\mathcal{B}^{\mu+(n+1)\times\delta})!I = \alpha(\mathcal{P}\mathcal{B}_t^{n+1})!I$

$$\begin{aligned} \alpha(\mathcal{B}^{\mu+(n+1)\times\delta})!I &= \alpha(\mathcal{B}^{\delta})!I[\alpha(B^{\mu+n\times\delta})] \quad \text{by decomposition} \\ &= \alpha(\mathcal{B}_t)!I[\alpha(\mathcal{P}\mathcal{B}_t^n)] \\ &= \alpha(\mathcal{P}\mathcal{B}_t^{n+1})!I \end{aligned}$$

□

The "only if" part (completeness) also hold. (Consider a number of iteration n and its successor $n + 1$.)

Once we know that H_1 holds, the satisfiability of H_3 reduces to the satisfiability of $\langle\mathcal{B} \mid \theta\rangle$. If $\langle\mathcal{B} \mid \theta\rangle$ holds, by induction, so does $\langle\mathcal{B}^{\mu+n\times\delta} \mid \theta\rangle$ for every n. Therefore, using lemma 9 of figure 6.3, assuming H_1 and $\langle\mathcal{B} \mid \theta\rangle$, H_3 holds.

6.5.2 The validator

Based on the results of section , we can define a validator V that enforces properties H_1, H_2 and H_3, therefore ensuring the validity of a run of the software pipeliner.

let V (i : `input_loop`, o : `output_loop`, θ : `observables`) =
$o.cond = i.cond = I <$ _
$\wedge$ `eq`$(\theta,\ \alpha(i.\mathcal{B}^{\mu}),\ \alpha(o.\mathcal{P}o.\mathcal{E}))$
$\wedge$ `eq`$(\theta,\ \alpha(o.\mathcal{B}_t o.\mathcal{E}),\ \alpha(o.\mathcal{E}i.\mathcal{B}^{\delta}))$
$\wedge\ \alpha(i.\mathcal{B}^{\mu})!I = \alpha(o.\mathcal{P})!I$
$\wedge\ \alpha(i.\mathcal{B}^{\delta})!I = \alpha(o.\mathcal{B}_t)!I$
$\wedge$ `dom`$(\theta,\ \alpha(i.\mathcal{B}))$
$\wedge\ Res(\alpha(B)!I) \subseteq I$

The correctness of the validator follows from theorems 6.3 and 6.4.

Theorem 6.5. *If* $V(i, o, \theta) =$ `true` *then*

$$\forall n, \alpha(\mathcal{B}^{\mu+n\times\delta}) \approx_\theta \alpha(\mathcal{P}\mathcal{B}_t^n\mathcal{E}) \ \ (H_1)$$

$$\forall n, \alpha(B^{\mu+n\times\delta})!I = \alpha(\mathcal{P}\mathcal{B}_t^n)!I \ \ (H_2)$$

$$\forall n, \langle\alpha(\mathcal{P}\mathcal{B}_t^n\mathcal{E}) \mid \theta\rangle \ \ (H_3).$$

6.6 Soundness of the symbolic model

The specification of the software pipeliner is the classical semantic preservation lemma. (The explanations use figure 6.1.) Suppose that the initial loop starts at point `in` in state S and executes down to point `out` in state S'. If the transformed code starts at point `in'` in T such that $S \cong_{theta} T$ we must build a sequence of reductions down to a state T' at point `out'` such that $S' \cong_\theta T'$. To prove this, we rely on the property enforced by our validator but the pipelined loop must also be used appropriately. In this section, we explain the properties that a correct use of the pipelined loop must set up and prove that if those properties hold, a successful run of the validator implies semantic preservation.

Let i be the original loop $\{c;\ \mathcal{B}\}$ and o be its pipelined version $\{c;\ \mathcal{P};\ \mathcal{B}_t;\ \mathcal{E};\ \mu;\ \delta\}$. For the semantic preservation lemma to be true, a few preconditions must hold. Some of them are syntactical.

Pre_1. N is invariant in i and o; N' is invariant in o;

Pre_2. I is incremented once in $\mathcal{B}$, μ times in $\mathcal{P}$, δ times in $\mathcal{B}_t$, and not at all in $\mathcal{E}$;

Pre_3. i is the sub-graph of the initial graph going from `in` down to `out`;

Pre_4.o is the sub-graph of the transformed graph going from `b` down to `c`.

Those conditions are easy to check and imply semantics properties that we will use through our development. Pre_1 implies that during execution of the loops, the bound is always equal to its original value. Pre_2 implies, for instance, that the value of the loop index I after $n + m$ executions of the loop body is equal to its value after n executions followed by m executions (recall that the value of the loop index is always bounded by the value of N so that there are no overflows). Pre_3 and Pre_4 states that we really are working on the loops that we are extracting from the graph and splicing back into.

We also need a few semantic preconditions:

Pre_5. $S.r(N)$ is greater or equal to μ;

Pre_6. $S.r(N) = T.r(N') + \rho(S.r(N))$;

Pre_7. The reduction from S to S' does not exit the loop and get back in.

Pre_5 is not necessarily true, we just restrict the study to case where it is, the other case being trivial (the control flows to a copy of the original loop). Pre_6 states that the bound was correctly computed. Pre_7 restricts the specification, without loss of generality, to the case where the execution did not get out of the loop to come back again. For a proof to be precise, this last precondition must be proved by carrying, in the semantic preservation invariant, the fact that the execution remains in the loop.

We can now state the specification of the software pipeliner. If the preconditions hold and assuming an execution of the initial loop from a state S to a state S' then, starting from an equivalent state T, the pipelined loop will execute to a state T' equivalent to S'.

Specification 6.1. *Assume that the preconditions Pre_1 to Pre_7 hold, that $S \cong_\theta T$ and $(\texttt{in}, S) \xrightarrow{*} (\texttt{out}, S')$. Then there exists a state T' such that $(\texttt{in}', T) \xrightarrow{*} (\texttt{out}', T')$ and $S' \cong_\theta T'$.*

The proof goes through two steps. The first separates the executions in their denotation and control components and is presented in the following section. Then, in section 6.6.2, using the soundness theorems of symbolic evaluation we show the correspondence between the symbolic model and the concrete execution.

6.6.1 Separation of denotation and control

The first reduction is based on the fact that conditions, in the Compcert semantics, do not modify the state. They only direct the flow of control. The key idea is, therefore, to decompose an execution into two components: on the one hand, a sequence of non-branching instructions have performed a state transformation; on the other hand, the various conditions have evaluated to some Boolean. The control decisions can be summarized by the evaluation of the conditions (the targets are irrelevant) and we write them down as $i < n$.

The following lemma states that if an execution goes through the loop, then three properties hold (it makes use of preconditions Pre_3 and Pre_7). First, the state transformation performed by the loop is equivalent to repeating $S.r(N)$ times the loop body (property P_1). Second, before the loop index reaches the value of the bound, all the conditions $I < N$ hold – and hence, the control flew into the loop body (property P_2). Third, when the value of the loop index reached the value of the bound, the condition $I < N$ does not hold any longer – and hence, the control flew out of the loop (property P_3). (Note that we implicitly use the preconditions: for instance, the bound is always expressed in state S).

Lemma 6.1. *If $(\texttt{in}, S) \xrightarrow{*} (\texttt{out}, S')$, then*

$$P_1. \mathcal{B}^{S.r(N)} : S \xrightarrow{*} S'$$

$$P_2. \forall k < S.r(N), \exists S_k, \mathcal{B}^k : S \xrightarrow{*} S_k \wedge S_k.r(I) < S.r(N)$$

$$P_3. \neg(S'.r(I) < S.r(N))$$

Dually, the following lemma (which makes use of precondition Pre_4) states that if a sequence of non-branching instructions performs some state transformation such that the evaluation of the conditions allows to execute exactly this sequence of instructions, then we can recast this state transformation into a concrete execution over the control-flow graph.

Lemma 6.2. *Assume*

Q_1. $\mathcal{PB}_t^{\kappa(T.r(N'))}\mathcal{E} : T \xrightarrow{*} T'$

Q_2.$\forall e < \kappa(T.r(N')), \forall T_e, \mathcal{PB}_t^e : T \xrightarrow{*} T_e \Rightarrow T_e.r(I) < T.r(N')$

Q_3.$\exists\ T_k,\ \mathcal{PB}_t^{\kappa(T.r(N'))} : T \xrightarrow{*} T_k \wedge \neg(T_k.r(i) < T.r(N'))$

Then $(\mathtt{b}, T) \xrightarrow{*} (\mathtt{c}, T')$

6.6.2 A symbolic model of the loops

Equipped with lemmas 6.1, 6.2, and the soundness theorems of symbolic evaluation, we can finally reduce the semantic preservation property that has to be enforced by the validator to a set of algebraic equations over symbolic states, the three properties presented in section 6.2.

As claimed, those three properties imply semantic preservation between the original loop and its pipelined version.

Lemma 6.3. *Assume that preconditions Pre_1 to Pre_7 hold and that*

$\forall n, \alpha(\mathcal{B}^{\mu+n\times\delta}) \approx_\theta \alpha(\mathcal{PB}_t^n\mathcal{E})\ \ (H_1)$

$\forall n, \langle\alpha(\mathcal{PB}_t^n\mathcal{E}) \mid \theta\rangle\ \ (H_2)$

$\forall n, \alpha(B^{\mu+n\times\delta})!I = \alpha(\mathcal{PB}_t^n)!I\ \ (H_3).$

If $S \cong_\theta T$ and $(\mathtt{in}, S) \xrightarrow{*} (\mathtt{out}, S')$ then there exists a state T' such that $(\mathtt{in}', T) \xrightarrow{*} (\mathtt{out}', T')$ and $S' \cong_\theta T'$.

Proof. In part (a) of figure 6.5 we present the synchronization schemes between the two loop executions, which is composed of three semantic preservation sub-proofs. The first and third are consequences of the preconditions. We focus on the second one, that is, we try to prove that there exists a state T_c such that there is a reduction from T_b and T_c is equivalent to S_i (the initial state reached by the original loop before it has to perform its last $\rho(S.r(N))$ iterations.)

Using lemma 6.1 we can decompose the execution of the initial loop between S and S_i. Therefore, we have, $\mathcal{B}^{S.r(N)} : S \xrightarrow{*} S_i$ (using P_1), $\forall k < S.r(N), \exists S_k, \mathcal{B}^k : S \xrightarrow{*} S_k \wedge S_k.r(I) < S.r(N)$ (P_2) and $\neg(S'.r(I) < S.r(N))$ (P_3).

Using theorem 6.1 with H_1, H_2 and P_1 we establish that there exists a state that we call T_c such that $\mathcal{PB}_t^{\kappa(T.r(N'))}\mathcal{E} : T \xrightarrow{*} T_c$ and $S' \cong_\theta T_c$. We have shown that the denotations of the loops are equivalent.

We must prove that the counters are synchronous. The synchronization schema is depicted in part (b) of figure 6.5. Let n be a positive integer less than $\kappa(T.r(N'))$ and T_n such that $\mathcal{PB}_t^n : T \xrightarrow{*} T_n$ (P_4). $\mu + n \times \delta$ is less than $\kappa(S.r(N))$. (Recall that $\kappa(S.r(N)) = \kappa(T.r(N'))$). By applying H_3 and P_2 and P_4 to theorem 6.2 we establish that $T_n.r(I) = S_{\mu+n\times\delta}.r(I)$ (P_5).

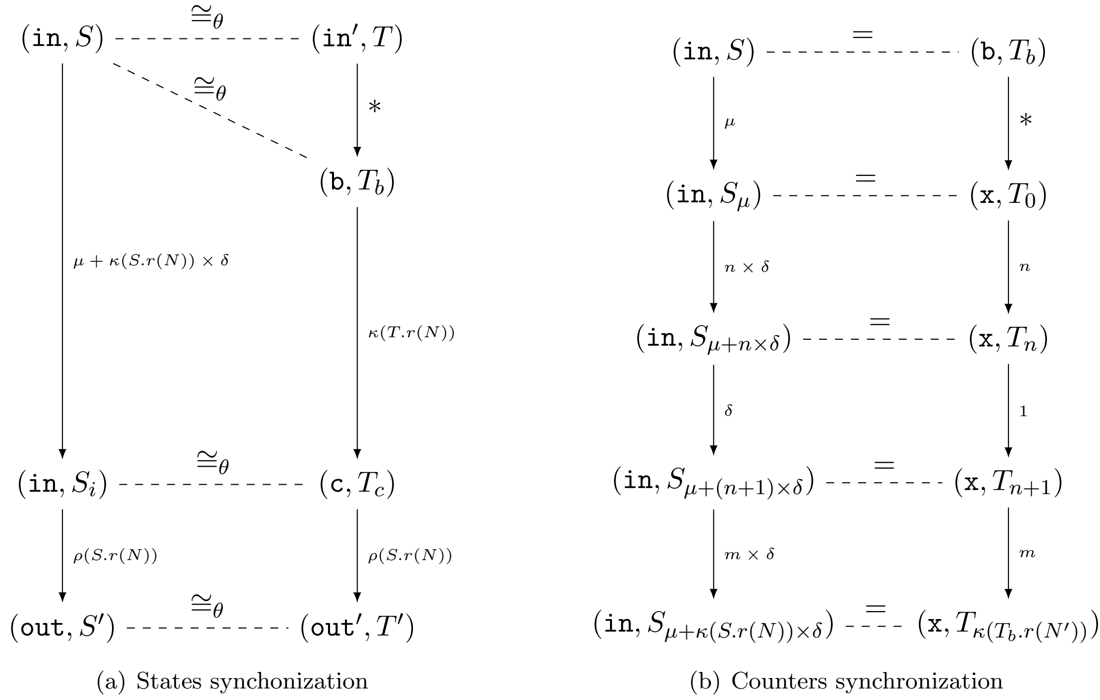


Figure 6.5: Synchronizations schemas

We then show that $T_n.r(I) < T_n.r(N)$ holds by relating its evaluation to the one in the original execution. $\mu + n \times \delta + \rho(S.r(N))$ is also less than $S.r(N)$. From P_2 we establish that $S_{\mu+n\times\delta+\rho(S.r(N))} < S.r(N)$. Then,

$$
\begin{array}{lll}
 & S_{\mu+n\times\delta+\rho(S.r(N))} < S.r(N) & \\
\Rightarrow & S_{\mu+n\times\delta} + \rho(S.r(N)) < S.r(N) & (Pre_2) \\
\Rightarrow & S_{\mu+n\times\delta} + \rho(S.r(N)) < T.r(N') + \rho(S.r(N)) & (Pre_6) \\
\Rightarrow & S_{\mu+n\times\delta}.r(N) < T.r(N') & (\rho(S.r(N)) \text{ less than both sides}) \\
\Rightarrow & T_n.r(I) < T_n.r(N') & (P_5, Pre_1)
\end{array}
$$

The same kind of reasoning applies in the case where $T_{\kappa(T_b.R(N'))}.r(I) < T_{\kappa(T_b.R(N'))}.r(N')$ does not hold.

We have established all the hypotheses of lemma 6.2, we use it to prove that the pipelined loop execution goes from (b, T_b) to (c, T_c) and we have already proved that $S_i \cong_\theta T_c$.

□

6.7 Discussion

6.7.1 Implementation and preliminary experiments

Our software pipeliner is implemented in OCaml and accounts for approximatively 2000 lines of Ocaml code. It uses a backtracking iterative modulo scheduler [App98] to produce a steady state. The modulo variable expansion follows the principles of Lam's expander of Lam [Lam88]. The modulo scheduler uses a heuristic function to decide the order in which to consider the instructions for scheduling. We tested our software pipeliner with two such heuristics. The first one randomly picks the nodes. We use it principally to stress the pipeliner and the validator. The second one picks the node using classical dependencies constraints. We also made a few schedules by hand. We experimented our pipeliner on the Compcert benchmark suite which contains a few numerical analysis programs such as a fast Fourier transform. Unfortunately, we did not observe any impressive speedups but there are at least three reasons for this. First, we do not use an alias analysis. Second, our heuristics are not state of the art in software pipelining. Three, we are running our programs on an out-of-order PowerPC processor. Using an in order DSP for instance would probably make a difference.

The validator has been implemented in OCaml and account for approximatively 300 lines of code. The validator was designed and implemented as we implemented the software pipelining. It is crafted to produce a lot of debug information, especially in the form of drawings of the symbolic states of the invariants. From a debugging point of view, the validator's debug information was an invaluable help, especially since the validation code is straightforward and small. For instance, we found many bugs in our preliminary implementation of modulo variable expansion. Figure 6.6 shows the drawings corresponding to the symbolic values of a software pipeliner run that introduced a semantics discrepancy.

Again, our implementation of symbolic evaluation is not as efficient as it could be because we did not implement hash consing of symbolic trees to perform sharing of sub-trees. The complexity of the symbolic evaluations required to check that the invariants hold is linear in the number of instructions and the comparison checking is exponential because of this lack of sharing but can be made linear with sharing. Nevertheless, in practice, as the loops that are chosen for software pipelining from our benchmark are innermost and never too big, it has not been a problem for our experiments. (Also, the extreme case in which trees grow exponential is rather unlikely to happen.)

All the underlying theory of symbolic evaluation, that is, sections 6.3 and 6.4, have been formalized in the Coq proof assistant. It accounts for approximatively 3500 lines of code. The proofs are big because symbolic states use Patricia trees and finite sets over symbolic values that are defined in a mutually recursive way. Nevertheless, the mechanization did not raise any unexpected difficulties.

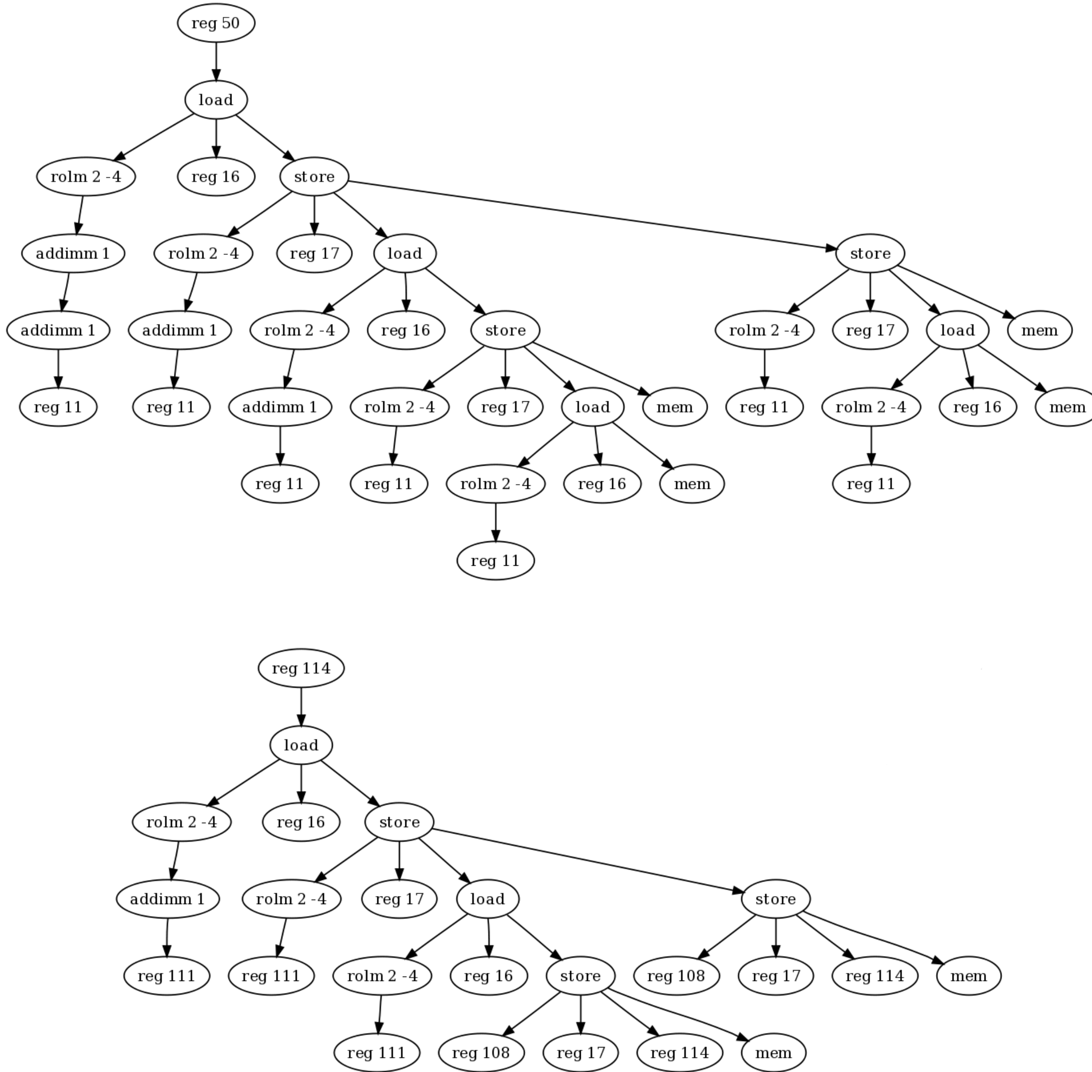


Figure 6.6: The symbolic values assigned to register 50 (top) and to its pipelined counterpart (bottom). The pipelined value lacks an iteration, due to a corner case error in the implementation of the modulo variable expansion. Symbolic evaluation was performed by omitting the prolog and epilog moves, thus showing the difference in registers use.

6.7.2 Related work

This work is not the first attempt to verify software pipelining. Leviathan and Pnueli [LP06] present a validator for a software pipeliner for the IA-64 architecture. The pipeliner makes use of rotating register file and predicate registers, but from a validation point of view, it is almost the same problem as the one studied in this chapter. Using symbolic evaluation, the validator generates a set of verification conditions that are discharged by a theorem prover. We chose to go further with symbolic evaluation and instead of using it to generate verification conditions, we use it to model the problem of semantic preservation for the software pipeliner. We claim that the resulting validator is simpler. Because symbolic evaluation hides all the syntactic details, we can express a single invariant for the loop denotations and one for the synchronizations of the loop indexes, which give an intuition of why the validators works. In the end, the validator is efficient, provably correct and with clear completeness conditions.

Another attempt at software pipelining verification is the one of Kundu et al. [KTL09]. It proceeds by parametrized translation validation. The software pipelining they consider is a code motion software pipelining, very different from the one of our study which is a modulo scheduling algorithm. This makes the validation problem rather different. In their case, the software pipeliner proceeds by rewriting the loop step by step, with each step being validated. In contrast modulo scheduling builds a pipelined loop in one shot (usually using a backtracking algorithm): it could possibly be viewed as a sequence of rewrites but this would be less efficient than our solution.

Another approach to software pipelining correctness is run-time validation as proposed by Goldberg et al [GCHP02]. The compilation pass takes advantages of the IA-64 architecture to instrument the software pipelined loop so that run-time check are performed to verify properties of the pipelined loop and recover from unexpected problems. This technique was used to verify that aliasing is not violated by instruction switching but they do not verify the full semantic preservation.

6.7.3 Conclusion

We presented a validator for software pipelining and its proof of correctness. The validation algorithm is significantly simpler than the software pipelining itself: it only performs a few symbolic checks while the pipeliner uses graphs algorithm such as the Lengauer and Tarjan algorithm and backtracking algorithms. The proof of correctness is also quite simple once the necessary infrastructure is set up. Therefore, this experiment is another strong example of the advantages of formally verified translation validation.

This experiment also shows how it is possible to use symbolic evaluation to reason about loop transformations, which is a novelty. Software pipelining has a very simple algebraic structure that makes it particularly interesting to reason about programs while hiding syntactic details.

We believe that this work shows that, even through symbolic evaluation is only a very primitive form of denotational semantics, it is perfectly suited to reason about advanced program transformations.

Chapter 7

Conclusion

7.1 Summary of the contributions

Our initial claim was that it is possible to use translation validation to build formally verified compiler passes, and that it is possible to design validators such that the proof of correctness is simpler than that of the transformation itself, the validator is insensitive to small variations of the heuristics or analyses used in the transformation, the validator does not raise false alarms and does not incur an unreasonable performance cost.

To evaluate this claim, we have presented four case studies: we considered four transformations drawn from the corpus of optimizations. In each of these case studies, we have designed a special-purpose validator and proved its correctness. These validators and their proofs are novel and constitute the primary contribution of this work. We believe that the validators satisfy our expectations. In each of the case studies, we compared the proof we built with what should have been formalized if the optimization itself was proved; it appears clearly that the data structure and algorithms we formalized for the validators are simpler. The three scheduling validators are insensitive to the heuristics used to build the schedule; the lazy code motion validator is insensitive to the dataflow analyses used to perform code motion. In each case, we studied the completeness of the validator, argued that they are complete for the kind of transformations we consider, and explained the conditions under which they should work. Finally, we paid attention to the algorithm and data structures used. Even though the complexity is not always optimal, the benchmark results were all satisfying. Finally, we have implemented the optimizations and the validators and, except in the case of software pipelining, we have plugged our verified compiler passes into the Compcert compiler.

This dissertation also contains additional technical contributions besides the validation algorithms and their proofs. It came to us as a surprise that ensuring that there are no spurious runtime errors in the transformed code was invariably among the most delicate problems. For the scheduling transformations, our solution is to extend symbolic evaluation with constraints.

For the lazy code motion validator, we used an anticipability checker.

Symbolic evaluation, despite its simplicity, turned out to be an extremely valuable tool to build validators. We extended it to handle traces, compare symbolic evaluations modulo observables, deal with runtime errors, studied its algebraic structure, and proved formally the fundamental theorem that relates symbolic evaluation with operational semantics. We believe that the use of symbolic evaluation to validate loop transformations is novel.

Finally, from a proof viewpoint, we have shown that semantic preservation invariants can be simplified by designing alternative program representations and semantics that encode properties about the program structure. We also demonstrated, with the lazy code motion proof design, how a proof of semantic preservation can be engineered such that it is easily reusable for other transformations.

7.2 Future work

There are several possible directions for improving formally verified translation validators. We have shown in chapter 5 that, with appropriate design and careful engineering, the lazy code motion validator can be reused for an unrelated optimization, namely constant propagation, with very few modifications. An interesting problem along this line is to design a formally verified general-purpose translation validator.

A possibility is to formalize existing general-purpose validators and prove their correctness. We have tried to formalize part of the framework proposed by Zuck et al. [ZPL01]. The validator is defined over an abstract semantic preservation invariant. This semantic preservation invariant is defined in terms of a relation between input and output codes nodes, a relation between the registers of both codes, an unknown invariant over the input code, and another one over the output code. A user who wants to use this generic validator for some untrusted optimization feeds the generic validator with the relations and invariants corresponding to the semantic preservation of the untrusted implementation. The result is a validator for the untrusted implementation. In the formally verified translation validation context, the user should get a proof of correctness of this validator along with the validator. Thus we need to make a generic proof of correctness of the generic validator. This is not completely straightforward. For instance, such a generic theorem needs hypothesis restricting the shape of the relations. One well known example is that the relation between the nodes must "cut" all the loops. Some of these hypothesis are difficult to formalize. Moreover it is not clear, when an hypothesis has to be made over the relations, whether it should be discharged by the user or validated, resulting in an increased compilation time. As a result, there exists a design space for such a proof and it is not clear what the good design is.

In such a framework, the user also has to provide the necessary invariants over the input code and output code. In the case of lazy code motion, a user would provides an available

expression analysis over the output code and a proof that the equalities hold for every executions. Unfortunately, there are many optimizations for which an available expression analysis is not sufficiently precise to validate the transformation. For instance, a validator using available expressions analysis to validate a redundancy elimination based on global value numbering would be incomplete. An interesting problem is to increase the precision of the static analysis that can be used by a generic validator. We have tried to complement our framework for lazy code motion with a global value numbering analysis over RTL code [GN04]. The formalization of this algorithm is difficult because it must use hash-consing to have a polynomial complexity.

Symbolic evaluation can be improved in two ways. First, we could perform hash-consing over symbolic evaluation trees. Indeed, we have shown in chapter 3 that the size of a symbolic evaluation expression can be exponential in the size of the sequence of instructions we evaluate when viewed as a tree but linear when viewed as DAG. A solution would be to have a hash consing library in the proof-assistant. Second, reasoning about symbolic evaluation could be facilitated by using decision procedures over symbolic formulas.

More pragmatically, formally verified translation validators may be improved in many different ways. An important feature if such a technology was deployed may be error diagnostics. The untrusted implementation may have to be tweaked by the end compiler-implementer who needs to understand whether the validator has detected a semantics discrepancy (and which one) in the untrusted implementation or raised a false alarm. The validation algorithm of Huang et al. [HCS06] is a very nice example of how a validator can be crafted to give informative error diagnostics. There are also some important optimization techniques that have not been addressed in this dissertation. One is alias analysis. Alias analysis is important because it can enhance greatly the quality of the optimization. Without alias analysis, memory operations block potential optimizations (for instance, two writes to memory cannot be switched, even when they write to distinct memory chunks). One way to look at this problem is presented in the work of Rinard and Marinov [RM99]. Another optimization technique that we have not addressed are interprocedural optimizations. Recent work from Pnueli and Zaks [PZ08] give insight on how to design such validators.

Finally, translation validation is just one form of validation focused on verifying semantic preservation. Ideas from formally verified translation validation may help in the construction of validators for other kinds of properties, such as validators used to enforce isolation as is done by Microsoft XFI [EAV+06] or Google's nativeClient [CYS+09].

7.3 Assessment

When I began working on the formal verification of translation validators, I did not have concrete evidence to justify the practical interest of this approach. Three years, four case studies, three publications, and a beautiful daughter later, the available evidence is now strong enough that I

am sometimes pleased to get the following question: can we formally verify the whole compiler by formally verified validation and therefore avoid to prove any of the compiler passes ?

In theory, the answer is probably yes. In practice, if we consider that the development of the formally verified compiler should be as simple as possible, my answer to the question is no: the best approach is to combine verified validation and verified transformation on a case-by-case basis, intelligently playing the strength of each approach.

On the one hand, as shown in this dissertation, there are program transformations for which verified validation is, I believe, the tool of choice. This includes for instance program transformations that move instructions around (like with scheduling or lazy code motion), or transformations such as register allocation, especially in the presence of sophisticated spilling strategies.

On the other hand, our experiments on constant propagation with strength reduction have shown that there are program transformations for which verified validation just seems to be an extra proof burden that also result in increased compilation time. I believe the same applies for transformations such as instruction selection and more generally local transformations that exploit algebraic identities between different instructions.

There are also transformations for which it is not clear what is the best choice, especially transformations that change the program's representation. We have designed a verified validator for such a transformation to deal with trace scheduling but it comes with one of the most technical proof of this dissertation. However, verifying this change of representation directly does not seem any less complicated.

In conclusion, formally verified translation validation is not the universal solution to all the problems raised by the verification of high-assurance compilers. Nonetheless, it appears as a formidable weapon in the arsenal of the formally verified compiler hacker.

Bibliography

[App98] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development – Coq'Art: The Calculus of Inductive Constructions.* EATCS Texts in Theoretical Computer Science. Springer, 2004.

[BFG+05] Clark W. Barrett, Yi Fang, Benjamin Goldberg, Ying Hu, Amir Pnueli, and Lenore Zuck. TVOC: A translation validator for optimizing compilers. In *Computer Aided Verification, 17th Int. Conf., CAV 2005*, volume 3576 of *Lecture Notes in Computer Science*, pages 291–295. Springer, 2005.

[BFPR06] Gilles Barthe, Julien Forest, David Pichardie, and Vlad Rusu. Defining and reasoning about recursive functions: a practical tool for the Coq proof assistant. In *Functional and Logic Programming, 8th Int. Symp., FLOPS 2006*, volume 3945 of *Lecture Notes in Computer Science*, pages 114–129. Springer, 2006.

[BGS98] Rastislav Bodík, Rajiv Gupta, and Mary Lou Soffa. Complete removal of redundant expressions. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 1–14. ACM, 1998.

[Coq08] Coq development team. The Coq proof assistant. Software and documentation available at http://coq.inria.fr/, 1989–2008.

[CYS+09] Brad Chen, Bennet Yee, David Sehr, Shiki Okasaka, Robert Muth, Greg Dardyk, Nicholas Fullagar, Neha Narula, and Tavis Ormandy. Native client: A sandbox for portable, untrusted x86 native code. In *2009 IEEE Symposium on Security and Privacy*. IEEE, 2009.

[Dav03] Maulik A. Dave. Compiler verification: a bibliography. *SIGSOFT Softw. Eng. Notes*, 28(6):2–2, 2003.

[Deu73] L. Peter Deutsch. *An interactive program verifier*. PhD thesis, University of California, Berkeley, 1973.

[EAV+06] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. Xfi: Software guards for system address spaces. In *USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 2006.

[Ell86] John R. Ellis. *Bulldog: a compiler for VLSI architectures*. ACM Doctoral Dissertation Awards. The MIT Press, 1986.

[GCHP02] Benjamin Goldberg, Emily Chapman, Chad Huneycutt, and Krishna Palem. Software bubbles: Using predication to compensate for aliasing in software pipelines. In *In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, page 211. IEEE computer society, 2002.

[Geu09] Herman Geuvers. Proof assistants: History, ideas and future. *Sadhana*, 34(1):3–25, 2009.

[GN04] Sumit Gulwani and George C. Necula. A polynomial-time algorithm for global value numbering. In *Static Analysis, 11th Int. Symp., SAS 2004*, volume 3148 of *Lecture Notes in Computer Science*, pages 212–227. Springer, August 2004.

[Gon08] Georges Gonthier. Formal proof – the four-color theorem. *Notices of the American Mathematical Society*, 55(11):1382–1393, 2008.

[Gul05] Sumit Gulwani. Program analysis using random interpretation. In *Ph.D. Dissertation, UC-Berkeley*, 2005.

[GZB05] Benjamin Goldberg, Lenore Zuck, and Clark Barrett. Into the loops: Practical issues in translation validation for optimizing compilers. In *Proc. Workshop Compiler Optimization Meets Compiler Verification (COCV 2004)*, volume 132 of *Electronic Notes in Theoretical Computer Science*, pages 53–71. Elsevier, 2005.

[Hal08] Thomas C. Hales. Formal proof. *Notices of the American Mathematical Society*, 55(11):1370–1380, 2008.

[Har08] John Harrison. Formal proof – theory and practice. *Notices of the American Mathematical Society*, 55(11):1395–1406, 2008.

[HCS06] Yuqiang Huang, Bruce R. Childers, and Mary Lou Soffa. Catching and identifying bugs in register allocation. In *Static Analysis, 13th Int. Symp., SAS 2006*, volume 4134 of *Lecture Notes in Computer Science*, pages 281–300. Springer, 2006.

[Huf93] Richard A. Huff. Lifetime-sensitive modulo scheduling. In *Proc. of the ACM SIGPLAN '93 Conf. on Programming Language Design and Implementation*, pages 258–267. ACM, 1993.

[HZ04] Mostafa Hagog and Ayal Zaks. Swing modulo scheduling for gcc. In *Proc. of the 2004 GCC summit*, pages 55–64, 2004.

[Jac09] Daniel Jackson. A direct path to dependable software. *Communications of the ACM*, 52(4):78–88, 2009.

[Kin69] James C. King. *A program verifier*. PhD thesis, Carnegie-Mellon University, 1969.

[KN03] Gerwin Klein and Tobias Nipkow. Verified bytecode verifiers. *Theoretical Computer Science*, 298(3):583–626, 2003.

[KN06] Gerwin Klein and Tobias Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. *ACM Transactions on Programming Languages and Systems*, 28(4):619–695, 2006.

[KRS92] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Lazy code motion. In *Programming Languages Design and Implementation 1992*, pages 224–234. ACM Press, 1992.

[KRS94] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Optimal code motion: Theory and practice. *ACM Transactions on Programming Languages and Systems*, 16(4):1117–1155, 1994.

[KSK06] Aditya Kanade, Amitabha Sanyal, and Uday Khedker. A PVS based framework for validating compiler optimizations. In *SEFM '06: Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods*, pages 108–117. IEEE Computer Society, 2006.

[KTL09] Sudipta Kundu, Zachary Tatlock, and Sorin Lerner. Proving optimizations correct using parameterized program equivalence. In *Proceedings of the 2009 Conference on Programming Language Design and Implementation (PLDI 2009)*. ACM, 2009.

[L+08] Xavier Leroy et al. The CompCert verified compiler. Development available at http://compcert.inria.fr, 2003–2008.

[Lam88] Monica Lam. Software pipelining: An effective scheduling technique for vliw machines. In *Proc. of the ACM SIGPLAN '88 Conf. on Programming Language Design and Implementation*, pages 318–328. ACM, 1988.

[LB08] Xavier Leroy and Sandrine Blazy. Formal verification of a C-like memory model and its uses for verifying program transformations. *Journal of Automated Reasoning*, 41(1):1–31, 2008.

[Ler06] Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *33rd symposium Principles of Programming Languages*, pages 42–54. ACM Press, 2006.

[Ler08] Xavier Leroy. A formally verified compiler back-end. Submitted, July 2008.

[LGAV96] Josep Llosa, Antonio González, Eduard Ayguadé, and Mateo Valero. Swing modulo scheduling: A lifetime-sensitive approach. In *IFIP WG10.3 Working Conference on Parallel Architectures and Compilation Techniques*, pages 80–86, 1996.

[LP06] Raya Leviathan and Amir Pnueli. Validating software pipelining optimizations. In *Int. Conf. On Compilers, Architecture, And Synthesis For Embedded Systems (CASES 2002)*, pages 280–287. ACM Press, 2006.

[LPP05] Dirk Leinenbach, Wolfgang Paul, and Elena Petrova. Towards the formal verification of a C0 compiler: Code generation and implementation correctness. In *Int. Conf. on Software Engineering and Formal Methods (SEFM 2005)*, pages 2–11. IEEE Computer Society Press, 2005.

[McC63] John McCarthy. Towards a mathematical theory of computation. In *Proceedings of the International Congress on Information Processing*, pages 21–28. C. M. Popplewell, 1963.

[MLG02] Josep M.Codina, Josep Llosa, and Antonio González. A comparative study of modulo scheduling techniques. In *Proc. of the 16th international conference on Supercomputing*, pages 97–106. ACM, 2002.

[Moo89] Strother Moore. A mechanically verified language implementation. *Journal of Automated Reasoning*, 5(4):461–492, 1989.

[MP67] John McCarthy and James Painter. Correctness of a compiler for arithmetic expressions. *Mathematical Aspects of Computer Science*, 1(19), 1967.

[MR79] Etienne Morel and Claude Renvoise. Global optimization by suppression of partial redundancies. *Communication of the ACM*, 22(2):96–103, 1979.

[Muc97] Steven S. Muchnick. *Advanced compiler design and implementation.* Morgan Kaufmann, 1997.

[MW72] Robin Milner and Richard Weyhrauch. Proving compiler correctness in a mechanized logic. In *Proceedings of the 7th Annual Machine Intelligence Workshop*, pages 51–72. Edinburgh University Press, 1972.

[Nec97] George C. Necula. Proof-carrying code. In *24th symposium Principles of Programming Languages*, pages 106–119. ACM Press, 1997.

[Nec00] George C. Necula. Translation validation for an optimizing compiler. In *Programming Language Design and Implementation 2000*, pages 83–95. ACM Press, 2000.

[PSS98a] Amir Pnueli, Ofer Shtrichman, and Michael Siegel. The code validation tool (CVT) – automatic verification of a compilation process. *International Journal on Software Tools for Technology Transfer*, 2:192–201, 1998.

[PSS98b] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In *Tools and Algorithms for Construction and Analysis of Systems, TACAS '98*, volume 1384 of *Lecture Notes in Computer Science*, pages 151–166. Springer, 1998.

[PZ08] Amir Pnueli and Anna Zaks. Validation of interprocedural optimization. In *Proc. Workshop Compiler Optimization Meets Compiler Verification (COCV 2008)*, Electronic Notes in Theoretical Computer Science. Elsevier, 2008.

[Rau96] B. Ramakrishna Rau. Iterative modulo scheduling. *The International Journal of Parallel Processing*, 24(1):1–102, 1996.

[Riv04] Xavier Rival. Symbolic transfer function-based approaches to certified compilation. In *31st symposium Principles of Programming Languages*, pages 1–13. ACM Press, 2004.

[RM99] Martin Rinard and Darko Marinov. Credible compilation with pointers. In *Workshop on Run-Time Result Verification*, 1999.

[RST92] B. Ramskrishna Rau, Michael S. Schlansker, and P. P. Timmalai. Code generation schema for modulo scheduled loops. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 158–169. ACM, 1992.

[Sam75] Hanan Samet. *Automatically Proving the Correctness of Translations Involving Optimized Code.* PhD thesis, Stanford University, 1975.

[SB99] Emin Gun Sirer and Brian N. Bershad. Testing Java virtual machines. In *Proc. Int. Conf. on Software Testing And Review*, 1999.

[Ste96] Bernhard Steffen. Property-oriented expansion. In *Static Analysis, Third International Symposium, SAS'96*, volume 1145 of *Lecture Notes in Computer Science*, pages 22–41. Springer, 1996.

[Str05] Martin Strecker. Compiler verification for C0. Technical report, Université Paul Sabatier, Toulouse, April 2005.

[TL08] Jean-Baptiste Tristan and Xavier Leroy. Formal verification of translation validators: A case study on instruction scheduling optimizations. In *35th symposium Principles of Programming Languages*, pages 17–27. ACM Press, 2008.

[TL09] Jean-Baptiste Tristan and Xavier Leroy. Verified validation of lazy code motion. In *ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation*, pages 316–326. ACM Press, 2009.

[TL10] Jean-Baptiste Tristan and Xavier Leroy. A simple, verified validator for software pipelining. In *37th symposium Principles of Programming Languages*. ACM Press, 2010.

[Wie08] Freek Wiedijk. Formal proof – getting started. *Notices of the American Mathematical Society*, 55(11):1408–1414, 2008.

[ZP08] Anna Zaks and Amir Pnueli. Covac: Compiler validation by program analysis of the cross-product. In *FM 2008: Formal Methods, 15th International Symposium on Formal Methods*, volume 5014 of *Lecture Notes in Computer Science*, pages 35–51. Springer, 2008.

[ZPFG03] Lenore Zuck, Amir Pnueli, Yi Fang, and Benjamin Goldberg. VOC: A methodology for translation validation of optimizing compilers. *Journal of Universal Computer Science*, 9(3):223–247, 2003.

[ZPL01] Lenore Zuck, Amir Pnueli, and Raya Leviathan. Validation of optimizing compilers. Technical Report MCS01-12, Weizmann institute of Science, 2001.

List of Figures

2.1 Architecture of the Compcert compiler. The thick arrows correspond to the compilation passes presented in this dissertation. 26

2.2 Semantics of RTL 33

2.3 Semantics of Mach instructions 37

2.4 Compiler construction. On the left, the usual compiler construction process of Compcert. On the right, the compiler construction with verified validators and untrusted implementation of optimizations. 40

4.1 The two extra rules of trace scheduling. On the left, an example of move after a condition. On the right, an example of move before a join point. On each example the code is shown before and after hijacking. 56

4.2 Overview of trace scheduling and its validation. Solid arrows represent code transformations and validations. Dashed arrows represent proofs of semantic preservation. 57

4.3 A code represented as a list of instructions (upper left), as a graph of instruction trees (upper right) and as a control-flow graph (lower left) along with its trees (lower right). 58

5.1 An example of lazy code motion transformation 70

5.2 Transfer function for available expressions 73

5.3 Effect of the transformation on the structure of the code 74

5.4 Three examples of incorrect code motion. Placing a computation of a/b at the program points marked by $\Rightarrow$ can potentially transform a well-defined execution into an erroneous one. 74

5.5 Anticipability checker 76

5.6 A few steps of the anticipability checking for computation $t_1 + t_2$. (Only the node at the way out of the graph holds the computation.) 77

6.1 High level oveview of a software pipelining transformation 91

6.2 An example of pipelined loop 93

6.3 Operators and basic properties of symbolic evaluation 98
6.4 The composition operator and its properties 102
6.5 Synchronizations schemas 110
6.6 The symbolic values assigned to register 50 (top) and to its pipelined counterpart (bottom). The pipelined value lacks an iteration, due to a corner case error in the implementation of the modulo variable expansion. Symbolic evaluation was performed by omitting the prolog and epilog moves, thus showing the difference in registers use. 112

List of Tables

1.1 Some the translation validation experiments 22

3.1 Sources of runtime errors. Many instructions are formalized as partial functions, and can therefore lead to a runtime error in the semantics. The last category of partiality source could be avoided by using a stronger type system. 45
3.2 Size of the development (in non-blank lines of code, without comments) 51
3.3 Compilation times and verification times 52

4.1 Size of the development (in non-blank lines of code, without comments) 67
4.2 Compilation times and verification times 67

5.1 Size of the development 84