B.Sc.Eng. Thesis
Bachelor of Science in Engineering

DTU Compute
Department of Applied Mathematics and Computer Science

A formally verified compiler back-end for a time-predictable processor

Frederik Krogsdal Jacobsen (s163949)

Kongens Lyngby 2019

DTU Compute
Department of Applied Mathematics and Computer Science
Technical University of Denmark

Richard Petersens Plads
Building 324
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk/english

Summary

For safety-critical systems, formal verification and time-predictability is needed to ensure functional safety. Formally verified compilers are needed to ensure correct translation from high-level programs to executable code, without which formal verification of the source program does not ensure safety of the system.
This thesis presents a partially implemented backend, which is designed to generate code for the time-predictable Patmos processor, for the formally verified compiler CompCert.
It is expected that the backend, if implemented completely, will enable higher safety of embedded systems.

Resumé

For sikkerhedskritiske systemer er formel verificering og tidsforudsigelighed nødvendigt for at sikre funktionel sikkerhed. Formelt verificerede oversættere er nødvendige for at sikre korrekt oversættelse fra højniveau programmer til eksekverbar kode, uden hvilken formel verificering af kildeprogrammet ikke kan sikre sikkerheden af systemet.
Denne afhandling præsenterer en delvist implementeret backend, som er designet til at generere kode til den tidsforudsigelige Patmos-processor, til den formelt verificerede oversætter CompCert.
Det forventes at backenden, hvis den implementeres fuldstændigt, vil gøre det muligt at konstruere sikrere indlejrede systemer.

Preface

This thesis was prepared at the Department of Applied Mathematics and Computer Science at the Technical University of Denmark in fulfillment of the requirements for acquiring a Bachelor of Science degree in Electrical Engineering.

The thesis work was supervised by Martin Schoeberl.

The reader is expected to be at least somewhat familiar with processors, compiler technology, and the foundations of mathematics. The thesis is structured to allow readers that are familiar with time-predictable processing, compilers, or mechanized proof to skip chapters 2, 3, or 4, respectively.

The implementation of the compiler backend can be found at `https://github.com/fkj/CompCert`. The implementation is not free software, but may be licensed for academic or personal purposes at no cost.

Kongens Lyngby, July 1, 2019

Frederik Krogsdal Jacobsen (s163949)

Contents

CHAPTER 1

Introduction

Many parts of modern life rely on computer systems to function. From medical devices, aircraft control systems, and nuclear power plants to stock trading machines, and even servers for web sites, computer malfunctions or failures have the potential of endangering human and animal life, causing massive economic losses, or substantially damaging the environment. Systems with this potential are called safety-critical systems.

To avoid failures in safety-critical systems, engineers typically follow a rigorous quality assurance program, which has traditionally included tests of all foreseeable inputs to the system, stress-testing of systems under load, and many other types of tests. Unfortunately, most systems are too complex to be exhaustively tested, and tests can thus never provide a complete guarantee that failures in the logic of the computer system will not occur. For this reason, standard and regulations, such as the ISO 26262 standard on functional safety for the electrical systems of road vehicles, prescribe that the functions with the most stringent safety requirements (e.g. brake-by-wire systems which may be the direct cause of deadly accidents if they fail) should be formally verified to ensure that the logic of the system is sound and implemented correctly [18].

To formally verify a system, the designer of the system must specify a set of requirements on the behaviour of the system. When the system is implemented, the system can then be proven to satisfy the requirements using a formal model of the requirements and a formal model of the system. In the past, such verification was an almost impossible undertaking, as all verification had to be done "by hand", but today, powerful computer tools allow for mechanized verification of most properties that might be specified in a requirement.

A problem with formal verification is that the system may still behave incorrectly, even if it satisfies the requirements, if the requirements themselves are not enough to guarantee safety. Additionally, if the implementation is translated in any way from the form it was in when it was proven to satisfy the requirements, the proof of correctness is no longer guaranteed to hold.

CompCert [Ler09b] is an optimizing compiler for a large subset of the C programming language which has been formally proven to correctly translate C code to assembly language for a variety of processors. This allows engineers to prove that high-level C code satisfies their requirements (which is typically relatively easy, compared to proving properties about assembly code), and then translate the high-level

code into assembly code that can actually be executed on a processor.

For many control systems, it is not enough for the system to make the right decisions. The system must also make the correct decisions within a set time frame if the system is to function correctly. For example, a brake-by-wire system must react very quickly when the brake pedal is pressed to avoid accidents.

It is very hard to determine how long it will take to execute a program on conventional computer processors, so specialised processors are needed to guarantee that systems will be fast enough for real-world programs. Processors which are designed to facilitate the determination of execution times are called time-predictable processors. Patmos [Sch+18] is a processor that is designed to make it easier to determine how long it will take to execute a program on the processor. Using Patmos and its accompanying tools, engineers can determine good bounds on the execution time of realistic programs, which makes it possible to guarantee that systems will be fast enough to be safe.

By combining the formally verified translation of high-level code to low-level code and the possibility of determining bounds on the execution time of programs, it becomes possible to improve the trustworthiness of safety-critical software development. However, there is, to the author's knowledge, no formally verified compiler for a time-predictable processor in existence today.

This project aims to develop and implement a backend for the CompCert compiler that allows it to translate C code into assembly code for the Patmos processor, thus enabling simultaneous formal verification of programs and practically applicable execution time analysis.

Having access to a toolchain that enables both formally verified, and thus functionally trustworthy, assembly language programs and execution time analysis may improve the safety of many classes of systems, especially those that have strict timing requirements and must not fail under any circumstances. However, the current level of technology will most likely still be too cumbersome and labour-intensive to apply generally, especially for systems that are not safety-critical.

Chapters 1 through 3 contain an overview of the theory required to understand, in broad terms, the workings of time-predictable processors, compilers, and mechanized proof. Each of these chapters may be skipped or quickly skimmed if the reader is already familiar with the topic in question. An overview of the synthesis of the two latter chapters, compiler verification, is presented in chapter 5, which also serves as an introduction to CompCert. The new work of the thesis is contained in chapters 6 and 7, which present the considerations of implementing the Patmos backend for CompCert, and an overview of the obtained results, respectively. Chapter 8 contains a review of related ongoing projects, while chapter 9 suggests future directions of research. Finally, chapter 10 contains the conclusion of the thesis.

CHAPTER 2

Time-predictable processing

One of the most important criteria when choosing a computer is the speed at which it can calculate things. The speed of calculation is normally limited by the speed of the computer's processor, but the speed of the memory, storage, and other peripherals may also factor into the equation, depending on which kinds of programs the processor must execute. Noting that the speed of the computer depends on the program that is executed, a sensible question is how the running time of a program can be calculated, and the answer may be surprising: in many cases, it is completely impractical to do so.

There are multiple reasons why it is often almost impossible to calculate the actual speed at which a program can be executed. First, both the processor itself and its peripherals may also be executing other programs, and these may be prioritised in a manner that is hard to determine in advance, especially as the computer receives user input, which may start and stop other programs. Fortunately, this is less of a problem for embedded systems, which often have a predictable set of programs executing at any given time.

It may also be the case that the peripherals of the processor are shared with other processors, so that the processor must sometimes wait to access e.g. the memory or a network controller.

Next, even if a processor is only executing a single program, the program itself may take many paths which have very different amounts of execution steps. This is complicated further by the fact that many modern processors speculatively execute branches, cache instructions and memory, and prefetch memory to lower the average execution time of programs executed on the processor. While these techniques can effectively lower the average execution time, they complicate the calculation of program running times significantly.

Finally, the program itself may be so complicated that it is impractical to determine the running time of all possible paths through the program. This problem can be mitigated somewhat by structuring the program to be more easily analysable, or by annotating the program to help the timing analysis tool.

2.1 Worst case execution times

Since it is often impractical to calculate the actual running time of a program, another approach is needed. Instead of calculating the precise execution time of each program, an approximation of the execution time can be used. The problem is then to find an approximation that is both useful, and easier to calculate than the actual execution time.

Not all processes have the same timing requirements. For example, a person using a personal computer will probably not mind—or even notice—if opening their web browser takes a second longer on some days than it does on others. Conversely, an airline pilot would probably mind it very much if the time between moving the control yoke and moving the airplane was suddenly increased by even a single second. There is a common element in these two examples, namely the concept of a sudden *increase* in processing time. It seems that for some processes, it does not matter if the processing time is sometimes longer, while for others, it does.

Consider now these similar, but slightly different examples. A person using a personal computer will probably mind if opening their web browser takes several minutes. Conversely, the captain of a large ship can accept that it takes several minutes for a manoeuvre to have any noticeable effect. Again, the two examples have a common element, but this time it is the *average* processing time.

From these examples, two elements of timing requirements spring forth: the *average* processing time and what is called the *worst-case* execution time, or WCET. For embedded systems, it is typically possible to specify a longest acceptable execution time, i.e. the longest time that a process may take without causing errors or failures. A time-predictable processor then, is a processor where it is "easy" to calculate the WCET of a process, so that it can be checked that it is less than the longest acceptable execution time for the process.

Unfortunately, this definition is a bit too strong to be practical, since it is typically not possible to precisely calculate the WCET of a process without knowing the exact inputs and conditions of the process. However, it is practically possible to calculate an upper bound on the WCET of a process. Additionally, some paths through a process may be infeasible, i.e impossible to reach no matter what inputs are given, and these paths may then be ignored in the analysis. This means that a more practical definition of a time-predictable processor is the following, which is due to Schoeberl [Sch09]:

> Under the assumption that only feasible execution paths are analysed, a time-predictable processor's WCET bound is equal or almost equal to the real WCET.

This definition separates time-predictable processors, which have good ("tight") WCET bounds, from normal processors, which typically have very conservative, overestimated WCET bounds since their designs incorporate techniques that optimise average execution time, but complicate WCET calculations.

2.2 The Patmos processor

Patmos is a time-predictable processor for embedded real-time systems [Sch+18]. It is designed to make worst case execution time analysis as easy as possible, while still retaining acceptable performance. Patmos is a dual-issue, statically scheduled RISC processor. Being statically scheduled, the processor does not reorder instructions, but relies on the programmer or compiler to schedule instructions for efficiency.

To enable time-analysable instruction caching, Patmos uses a method cache [Deg+14], which guarantees that all instructions except calls and returns result in cache hits when loading the next instruction. This simplifies the analysis of function execution times, as the instruction load times are thus more predictable.

2.2.1 A quick overview of the Patmos architecture

Patmos is a dual-issue processor, so instructions can be bundled to be executed in parallel. The bundles can either be 32 bits or 64 bits wide, depending on whether the bundle contains one or two instructions. All instructions are 32 bits wide, but some load instructions can load an immediate 32-bit value from the second instruction slot in the bundle. All instructions have constant execution time, provided they do not miss caches, but some instructions need delays slots in the pipeline to finish execution. Control flow instructions and memory accesses can only be scheduled in the first instruction slot of a bundle, but arithmetic and logic instructions can be scheduled in any instruction slot.

In contrast to many common RISC architectures, all Patmos instructions are predicated, which makes it possible to reduce the number of branches needed in a program, and enables single-path programs. Patmos has 8 predicate registers which can be set using compare instructions and used to predicate instructions.

The register file of Patmos contains 32 general purpose 32-bit registers, the 8 one-bit predicate registers, and 16 32-bit special purpose registers. The special purpose registers include multiplication result registers, spill and stack pointers, and return address registers. The multiplication result registers are needed because the multiplication instruction of Patmos is executed in a separate pipeline, and thus needs a delay slot in the pipeline before the result is available.

The Patmos architecture does not contain any instructions or registers for floating point numbers or 64-bit integers.

Patmos contains three caches and two scratch-pad memories for data and instructions. The instruction set has typed stores and loads, allowing fine-grained control over memory accesses. The caches include the previously described method cache, a stack cache and a data cache.

Patmos is optimized for single-threaded performance, but many Patmos cores can be integrated into the T-CREST project as nodes in a multi-core system for multi-threaded processing [Sch+15]. The Patmos cores in such a system share the main memory via a memory arbiter.

2.2.2 The existing toolchain

The Patmos processor is not currently realised in hardware. However, there exist both a simulator and a cycle-accurate emulator for the processor. It is also possible to synthesise the processor design for a range of common field programmable gate arrays.

The only existing compiler for Patmos is a port of the LLVM compiler for the C language. The compiler is an important part of ensuring a good WCET bound, as it generates information used by the WCET analysis tool [Pus+13].

CHAPTER 3

Compilers

A compiler is essentially a program that translates high-level source code, which can be understood by humans, into low-level machine code, which can be understood by computers. While low-level machine code can also be understood humans, it is not very easy to read or write large programs in, since the steps taken by each instruction are very small. In a high-level language, one might be able to write `x = 2 * x`, but in a low-level language, much more detail must be provided. What, for example, does `x` mean? Intuitively, one might say that `x` is a variable, i.e. a place that some value can be stored, but in a low-level language, one must also specify what kind of value `x` is, and where exactly it is to be stored. Thus, in a low-level language, `x = 2 * x` may translate into a sequence of instructions similar to the following:

1. Retrieve the current value at storage location `x` as a signed 32-bit integer, and store it inside the processor at register location 1.
2. Multiply the value at register location 1 by two, and store the result in register location 2.
3. Retrieve the value at register location 2 and store it at storage location `x` as a signed 32-bit integer.

Reading this, one easily sees why it is impractical to use the low-level language for applications of any notable size: the programmer is forced to take into consideration all sorts of information that is not relevant to the matter at hand, and which obfuscates the actual meaning of the program. Instead, most programmers choose to program in a high-level language and then compile their code to a low-level language, which means that good compilers are an essential requirement for any processor to be practically useful.

3.1 Overview of the phases of a compiler

The phases of a compiler can be very broadly split in two parts: analysis and synthesis. The analysis phases break the source code into parts, determine what each part means, and create an intermediate representation of the source program. The synthesis phases then operate on this intermediate representation, finally constructing a program in the target language. In general, analysis is not very hard compared

to synthesis, as most programming languages are designed to be easy for computers to understand (in opposition to natural languages, which are typically very hard for computers to understand). Instead of categorising the phases in analysis and synthesis parts, the compiler may also be split in a front-end and a back-end. The front-end part then consists of those phases that primarily depend on the source language, while the back-end consists of those phases that primarily depend on the target language. There may be some phases which are hard to place exclusively in the front-end or the back-end, especially the phases in the "middle" of the compiler, which typically do not depend on either the source or the target language, but only on the compiler's internal intermediate representation of the program [ASU86].

The phases of most compilers are organised in more or less the following sequence [ASU86] [App98]:

1. Lexical analysis
2. Syntax analysis
3. Semantic analysis
4. Intermediate code generation
5. Optimisation phases
6. Target code generation

The rest of this section will detail what these phases involve.

A simple compiler can forgo the optimisation phases, and may not have a "formal" intermediate code generation phase, but may simply translate the syntax of the source language more or less directly into the syntax of the target language. The main reason for splitting the compiler phases into a front-end and a back-end is to make it easier to re-target the compiler, i.e. to create either a new front-end or a new back-end and couple it to the existing compiler, thus saving a large part of the work needed to create a compiler from scratch.

3.1.1 Lexical analysis

The lexical analyser reads the characters of the source program and produces from them a sequence of "tokens". These tokens are the basic elements of the source programming language such as keywords, operators, literal values, constants, variable identifiers, and so on. The tokens can be thought of as the "words" of the source program, and following this analogy, the lexical analyser takes a sequence of "letters" and turns it into a sequence of "words".

Since the lexical analyser is the part of the compiler that interacts with the characters of the source program, it is typically also responsible for several auxiliary features

such as removing comments and white space from the source code, gathering information about token positions for error reporting, and some macro expansions if the source language supports these features.

The lexical analyser turns the source characters into tokens by recognizing certain patterns in the character sequence. These patterns define a set of strings that are associated with the token. A string that is in this set is called a lexeme. For example, in the C statement `int length = 34;`, the character sequence `length` matches the pattern for identifier tokens. Thus the lexical analyser will turn the character sequence into an identifier token, with the lexeme being the string "length". Similarly, the character sequence 34 will be matched as a literal numeric token with the lexeme being the string "34".

The patterns are typically defined using regular expressions. These may then be automatically turned into finite automata, which can be efficiently implemented as compressed transition tables [ASU86].

3.1.2 Syntax analysis

Once the lexical analyser has turned the source program into a sequence of tokens, the compiler must determine the syntactic meaning of the token sequence. For example, a C program consists of blocks, which consist of statements, which consist of expressions, which consist of tokens. The compiler must not only check that the source program follows the syntactic rules of the language, but also construct a data structure that represents the syntax of the program. The syntax analyser (also called the parser) performs this task.

The syntax of a programming language is typically specified using a context-free grammar. From a context-free grammar, it is possible to automatically construct efficient syntax analysers. The grammar determines which sequences of tokens are valid programs and which are not. Additionally, the grammar can be used to impose associativity and precedence on operators, so that expressions can be grouped systematically without the need for the programmer to manually assign groupings to each token.

The syntax of the program can be represented as a parse tree, which describes how the sequence of tokens "fit together" into expressions, statements, and so on. If the parser encounters a sequence of tokens that fits some pattern in the grammar, it will add a new branch to the parse tree to contain the construct that the tokens represent. On the other hand, the parser may also encounter token sequences which match no pattern in the grammar. In this case, the parser will typically implement an error system which can tell the programmer where and why problems were encountered using the information gathered by the lexical analyser.

There are several ways to implement a parser. The methods most commonly used in production compilers are classified as either bottom-up or top-down. Bottom-up parsers start from the "leaves" (i.e. the individual tokens) of the parse tree and work up to the root (i.e. the entire program), while top-down parsers start from the root and

work down to the leaves of the parse tree. In addition to these two categories, there are several ways to define the grammar and parser to support different functionalities and gain performance in certain cases.

3.1.3 Semantic analysis

Once the compiler has created a parse tree from the source program, it has determined that the sequence of characters in the source program can be made to fit into the grammatical rules of the source programming language. However, it still has no concept of the actual meaning of the program, and it may even be that the program has no meaning at all. In the same way as the grammar of a natural language such as English may permit meaningless sentences that are grammatically correct, the grammars of programming languages also often permit programs which can be assigned no meaning.

Typically, meaning is lost due to missing name definitions and operations on data types for which the operation is not defined. Thus the compiler must check that each name referred to in the source program is actually defined, and that the type of each expression is correct. For some languages, the compiler may also defer the type check until the program is actually executed, which is especially useful for dynamically interpreted languages.

In summary, when the compiler has determined that the source program *can* have meaning, it must determine the actual semantic meaning of the program. This is often accomplished by attaching semantic actions to each grammar production during the parsing of the source program. Once the parse tree has been constructed, including these semantic actions, it may be traversed to evaluate the semantic actions. Semantic actions may generate code, modify the list of names, generate errors, and so on. By traversing the parse tree and evaluating the semantic actions, the compiler may thus translate the source program into a mathematical representation of the meaning of the program.

The mathematical representation is in many cases simply another language, and it may even be the target language of the compiler. Most often, the compiler will first translate the source program to an intermediate language which is suitable for optimisations. Some compilers, like CompCert, will even go through several intermediate languages, each of them suitable for specific manipulations of the original program.

3.1.4 Intermediate code manipulation

Once the compiler has translated the original program into an intermediate language suitable for manipulations, it can begin applying different operations to the program. Depending on the goals of the compiler, it may optimise and instrument the code in different ways. Most industrially used compilers will attempt to optimise the code using various methods, which will be explained in section 3.5. It is often possible for

the programmer to select whether the compiler should optimise the code for speed or for memory usage, or perhaps for other considerations.

Some compilers will also add instrumentation that can make further analysis of the program in the target language possible. One example of this is compilers which can add code annotations for worst-case execution time analysis.

3.1.5 Target code generation

The final phase of the compiler is target code generation. It turns the intermediate language program into a program in the target language. In simple compilers that do not optimise the code, this phase may come directly after, or even be a part of, the semantic analysis phase.

The target code generator must of course produce correct code in the target language, but it should also produce effective code, and do it reasonably quickly, to be practically usable. It is also practical for the compiler to be able to produce relocatable target code, so libraries and programs can be compiled separately and linked together after compilation.

The target language of practical compilers will often be assembly language for the target machine. This introduces the need for an assembly step before program parts can be linked together, but makes the target language more readable and possible to modify manually compared to choosing machine language as the target language.

One of the main problems of target code generation is determining which machine instructions can be used to implement each expression of the program in the intermediate language. The difficulty of selecting instructions for an expression depends greatly on the instruction set of the target machine, as some instruction sets may have operations that are very close to the operations of the intermediate language, while others may require several instructions to emulate a single operation of the intermediate language.

Another important consideration is how to store the data of the program. Real machines have only a limited amount of registers in the central processing unit, so the compiler must decide what data to store in the registers, and what data to store in the slower memory. Additionally, the compiler must decide how to partition the memory so that there is space for the program code itself, any statically allocated data, and any dynamically allocated data. It may also be that the intermediate language can represent data that can not be stored in a single register on the real machine, e.g. 64-bit numbers in a program for a machine that has only 32-bit registers. In this case, the compiler must decide on a system for working with this data.

3.2 Type checking

A very useful compiler feature is the ability to type check the source program. Type checking is a static check (i.e. it happens while compiling the program, not while the

program is running), that determines whether operators and functions are applied to operands that make sense. For example, it does not make sense to apply the conventional arithmetic multiplication operator to two string values. To the machine, however, string values are also simply numbers, so most machines will happily attempt to "multiply" two string values, in most cases producing nonsensical results. This can make it very hard to find the source of such errors in the program without the help of a compiler with type checking.

To facilitate this analysis, each construct of the programming language must have a defined type. Some operator constructs may have different types depending on the type of the operands—this feature is called overloading. For example, the arithmetic operators of a language can often be applied to both integers and floating point numbers.

Types can often be divided in the basic, or primitive, types such as integers, characters, and floating point numbers, and constructed types such as arrays, enumerated types, and various other structures depending on the language. In the C language, for example, pointers and functions can be considered constructed types. The limits of what can be done with constructed types depend heavily on the source programming language. For example, some languages allow functions to take functions as arguments and return functions; these languages are called (higher-order) functional programming languages.

Types can be determined during the syntax analysis by adding type rules to the grammar of the language. It is also in some cases possible to infer the type of a construct from contextual information. For example, it is possible in many languages to write statements like `height` = 4, from which the compiler can automatically infer that the variable `height` should have an integer type. The compiler can then report an error if the variable is assigned a value of another type later in the program.

One of the main problems of type checking is deciding whether two types are equivalent. Deciding equality between basic types is of course easy, but problems arise when types are constructed from other types or given shorthand names (aliases). Then situations may arise where several types exist that are actually identical by definition, but which have different names. This problem can be solved by various unification algorithms that attempt to substitute in definitions for constructed types and aliases to make two types equal.

3.3 Run-time environment

Before target code can be generated, the compiler must determine the relation between names in the source program and data constructs (memory locations). As the program is executed, the same name can refer to several pieces of data, since it is typically possible to e.g. declare local variables with the same name as a global variable due to scoping rules.

Allocation and deallocation of data constructs in the memory is managed by a run-time environment, which consist of auxiliary functions that are loaded with the

generated program code. The way a data construct is stored in memory depends on the type of the data. The basic data types can typically be stored "as-is" in memory, while constructed data types must be represented as collections of basic constructs. The structure of these collections depends on the type of constructs implemented by the source and target languages.

Each name in the program that is allocated in memory is collected in a symbol table, which the compiler uses to determine the location of the data associated with the name. Whenever a name is defined (or an existing name is changed) in the source program, the symbol table must be updated. Thus the symbol table is often implemented as a hash table for efficiency.

For many languages, whenever the name of a function appears in the source program, the function is called at that point. When a function is called, the computer will execute the function body. At the definition of the function, the type and amount of arguments of the function will typically be defined using special identifiers. These identifiers are called the formal parameters of the function. When the function is called, the programmer must supply values for each of these identifiers. These values will be substituted into the function definition before it is executed—they are then called the actual parameters of the function.

Every call of a function is called an activation of the function. Since functions can (in many languages) be recursive, it is possible for several activations of a function to be active at the same time. Fortunately, it is possible to organize the function activations of a program as a tree, considering the program itself as a function, which is the root node of the tree. The control flow of the program can be described as a depth-first traversal of the activation tree. This can be represented as a stack (the control stack), which contains all active activation nodes in the order they were activated. The control stack can then be used to determine where to go once the current activation is finished.

The compiler may divide the memory into three overall sections: one for the target code itself, one for data constructs from the program, and one for the control stack. The first part contains the list of instructions of the program, which has a fixed size, so the compiler can calculate the size of this section at compile time. The second part contains both statically and dynamically allocated data constructs from the program itself. The size of the statically allocated constructs can also be calculated at compile time, so they can be placed in a memory area of fixed size. The third part contains no statically allocated data, since the data is only saved when a function is actually called.

Dynamically allocated data is placed in a memory area called the heap. Since both the heap and the stack can change sizes, it is usual to place them at opposite ends of the memory space so that they can grow towards each other. It is of course critical that the heap and stack do not grow so much that they "collide". When this happens, it is called either a stack overflow or a heap overflow and it typically causes the program behave incorrectly or "crash", i.e. stop functioning altogether. Compilers can not prevent overflows, but they may have mechanisms to detect them and stop the program if they happen to prevent incorrect behaviour or security issues.

3.3.1 Activation records

The information needed to process a function call includes the actual parameters, the location of the function code in the memory, and the memory location and machine state of the function call, so that the processor can return there once the function has finished executing. Additionally, some memory may be needed to store temporary values and local data that is needed during the execution of the function, as well as the return value of the function. In practice, the actual parameters and return value of the function are often stored in predefined machine registers instead of memory for faster execution time. All of this data is stored in a block of storage called the activation record (or frame) of the function call. The activation records of the program are stored on the control stack, organized according to the activation tree as described above.

If function calls are nested, it may be necessary to also store an "access link" to the activation record of the calling function so that the function can access values of variables that are local to the calling function. This pattern may continue if there are multiple nested function calls, each subfunction using the access link of its caller to access the next function in the stack.

Most of the data sizes in an activation record can be computed at compile time. The most common exceptions concern local data with constructed data types with sizes that depend on actual parameters, e.g. arrays with lengths that depend on parameters. For this reason, the location of the local data in an activation record can be accessed by adding a known offset to the absolute location of the start of the activation record.

To construct and destroy the activation record, the compiler must insert additional code before and after each function call. These pieces of code are called the call sequence and the return sequence. The call sequence evaluates the actual parameters, stores a return address and the previous location of the top of the stack, moves the top of the stack up to make space for the new activation record, saves the machine status, initializes local variables, and finally jumps to the start of the function. The return sequence places the return value of the function in the activation record of the caller, restores the saved machine state, moves the top of the stack down to its previous location, and jumps to the saved return address.

3.4 Instruction selection

Intermediate languages typically express as few operations in each statement as possible to simplify the manipulations done in the optimisation passes. But a real machine instruction can typically do several things at once. Thus it can be very challenging to find an efficient implementation of the intermediate language program in real machine instructions. In fact, the problem of generating optimal code is undecidable [ASU86].

If efficiency is of no concern, instruction selection can be implemented using a very simple algorithm: simply determine a sequence of machine instructions that implements each intermediate language operation, and replace each intermediate language operation with that sequence. Unfortunately, such an algorithm will often produce very inefficient code. This is because looking at each operation individually will often produce redundant code or several instructions that could have been combined into a single instruction.

Re-targetable code generators can be efficiently implemented as tree-tiling algorithms. The basic idea is to parse the intermediate language program into a parse tree (unless it is already represented as one internally). Each machine instruction can then be expressed as a fragment of an intermediate language tree, which will be called a tree pattern. The problem is then to cover the parse tree with tree patterns as cheaply as possible. The exact definition of "cheapness" depends on both the target machine and the optimisation goals, but for reduced instruction set computers, the cost function typically ends up being simply the amount of instructions used, since most instructions for such machines have almost equal costs and functionality.

3.5 Common optimizations

For compilers for industrial use, simply translating the source program into the target language is not enough. The compiler must also translate the source program into an efficient program in the target language. It is often even possible for the compiler to optimize the source program itself by removing inefficiencies introduced by the programmer. For example, most modern compilers remove unused variables and unnecessary assignments. This section will explain some of the most common compiler optimizations, including most of the optimizations implemented in the CompCert compiler.

To implement optimizations, the compiler will often need to determine when the program may transfer control flow to another part of the program. Each sequence of statements in which control flow can not be transferred is called a basic block of the program. The basic blocks of a program can be collected in a control flow graph to show how control can transfer throughout the program. Local optimizations are those that can be implemented by inspecting only the statements in a single basic block, while global optimizations require information from several basic blocks. Many optimizations can be performed at both local and global levels.

3.5.1 Dataflow analysis

An important class of optimizations are those that can be implemented using dataflow analysis. A dataflow analysis is performed by traversing the control flow graph and gathering information about which situations may arise during execution. Naturally, a dataflow analysis will almost always be only a suboptimal approximation, as it is impractical to check all possible situations.

An example of an optimization based on dataflow analysis is constant propagation. Constant propagation works by substituting the values of constants (i.e. variables) that are known at compile time into the expressions that depend on the constants. If other variables depend on the constant, this process may also make the other variables effective constants. Thus the constant propagation process may make some variable declarations redundant, even if they originally depended on other variables.

Another example is common subexpression elimination. A common subexpression is an expression that has been computed previously in the program, and in which the variables of the expression have not changed since it was last computed. The program can avoid recomputing a common subexpression by storing the computed value until it is needed again, and then using it directly instead of computing it again.

A variable is called *live* at a point in the program if its value is used past that point. If its value is not used past that point, the variable is called *dead* from that point on. Code that serves only to compute dead variables is called dead code. Dead code elimination optimizes the program by eliminating dead code. At first glance this optimization may not seem very useful, as most one might expect most programmers to rarely write useless code. However, other optimizations, such as constant propagation, will often produce dead code by removing the expressions that depend on the code in question.

For embedded systems, peripherals may often be memory mapped to specific locations in the memory. Pointers to these locations may look dead from the point of view of the compiler if the program itself never uses the values stored there, or the pointers may look to be constants if the program never writes any values to the location. However, the locations may be used for various critical functions, such that overzealous constant propagation and dead code elimination may break the program functionality. For this reason, many compilers allow the programmer to annotate variables as “volatile”, meaning that they have meaning outside of the program and should not be optimized away.

3.5.1.1 Register allocation

In the intermediate language programs, and sometimes even during code generation, compilers generally assume that an infinite number of registers are available to the processor. Before the final target language program can be generated, the compiler must decide which variables will actually be stored in registers, and which registers will hold which variables. Unfortunately, finding an optimal register assignment is an NP-complete problem [ASU86], so most assignment algorithms are heuristic in nature.

A common way to allocate registers is to construct an interference graph from the control and dataflow graphs. Two values in the intermediate program are said to interfere if they can not both be stored in the same register. One of the most common reasons for two values to interfere is that they are both live at the same time, but it is not the only one. It may also be that a register is needed to store return values or other information for function calls. Finally, instruction set restrictions on which

registers can be used with certain operations can be encoded as interference between the result registers of the operations and the registers that can not be used with the operations.

The interference graph has each value in the program as nodes and edges between values that interfere. To allocate registers, the interference graph must now be coloured so that no nodes with single edges between them have the same colour, using as few colours as possible. This will ensure that no interfering values share a register. Of course, there are only a limited number of registers, and so only a limited number of colours. If the graph can not be coloured without using more colours than there are registers, the compiler will have to move some of the values to memory, which will be slower than using registers. The process of moving values to memory is called *spilling* the values.

Another optimization made possible by the interference graph is register coalescing, which is the process of removing unnecessary move instructions. If there is no edge between the source and destination registers of a move instruction, it is not necessary to move the value, and the move instruction can be removed, and the source and destination nodes are merged into a single node with the edges of both nodes. Some care must be taken not to increase the number of colours needed to colour the interference graph when coalescing, as the merged node is more constrained than the previous two nodes. If registers are recklessly coalesced, the graph may become so constrained that the compiler needs to spill values, which is typically much more expensive than simply not coalescing the registers in the first place. There exist algorithms that allocate and coalesce registers without introducing unnecessary spills, e.g. optimistic colouring [BCT94] or iterated register coalescing [GA96].

3.5.2 Tail call optimization

A function call is said to be a *tail call* if it is the last instruction that will be executed before returning from the enclosing function. For example, `f(x − 1)` and `g(x)` are tail calls in the following function `f(x)`, while `h(x)` is not a tail call because the addition instruction must be executed after the function call:

```
int f(int x) {
  if (x > 0)  return f(x − 1);
  if (x < 0)  return g(x);
  return 1 + h(x);
}
```

When a tail call is encountered, the compiler may optimize the generated return sequence by jumping directly from the end of the tail called function to the function that called the enclosing function. In the example above, once the function call `g(x)` has been executed, control may return directly to whatever code called the function `f` in the first place instead of first returning to `f`, then returning again immediately.

Recursive tail calls are especially useful because they do not need multiple activation records (since the compiler can simply update the existing activation record and

jump to the start of the function). This makes recursion take much less stack space than otherwise needed.

3.5.3 Function inlining

If a program calls a function, it may be more efficient for the compiler to simply copy the function into the place it is used, so that the call and return sequences do not have to be generated. This idea is called function inlining, or inline expansion. If all calls to a function are inlined, the function itself will never be called, and may thus be eliminated from the program.

A danger of function inlining is that local variables in the function body may have the same name as variables in the scope of the function call. Thus simply copying the function body may not be enough. This danger can be avoided by either renaming the problematic variables, or by simply renaming all variables in the program so that no variables have the same name before starting the optimization pass.

Another danger is that function inlining may not always be more efficient than simply leaving the function as is. It is in fact possible to construct programs where indiscriminate function inlining may make the program infinitely long [App98]. There are several heuristics that can be used to determine whether function inlining is worthwhile. Generally, functions that are smaller than the call and return sequences, and functions that are called only once are worth inlining, as this will always save code size. Functions that are called in deeply nested loops or functions that are found to be frequently executed via profiling may also be worth inlining. Some compilers allow the programmer to give a hint to the compiler that a function may be worth inlining by annotating the function.

3.6 Preservation of semantics

Most of this chapter has focused on how compilers work in general terms, as well as how a compiler may optimize the source program for speed and/or memory usage. However, one of the most important properties of a compiler has been neglected until now, namely the problem of whether the generated program in the target language actually has the same meaning as the source program. If it does, the compiler is said to preserve the semantics of the source program. It should be obvious that any compiler *should* preserve the semantics of the source program if it is to be useful. Unfortunately, several studies have found that *none* of the most popular compilers for the C language preserve the semantics of all possible source programs [Yan+11] [ER08].

These “wrong-code errors” are theorised to happen primarily because the optimizations mentioned previously in this chapter are not always implemented correctly. This can be because an optimization is applied with incorrect assumptions, because the theory of the optimization is flawed (i.e. the optimization does not preserve the

semantics of the code, even in theory), or because the code implementing the optimization is simply wrong. However, the parsing and generation of code itself may also be flawed.

To combat these errors, compilers can of course be tested on large test suites of programs, but as the set of valid programs is infinite for most programming languages, this can never guarantee that the compiler is correct. It would be better if it was possible to prove mathematically that each part of the compiler was correct. However, such a theorem would be almost impossible to prove by hand, as compilers are often very large programs. To see how it is possible to prove very large theorems in a manageable way, it is necessary to introduce the concept of mechanised proof.

CHAPTER 4

Mechanised proof

When using mathematics to solve a problem, one will almost always use some theorem or result that does not seem obvious, but is typically accepted without any further thought. The reason mathematicians (and engineers) rest easy despite doing this, is that they know that the theorems they use have been *proven* correct by someone else. For instance, not many people would object to the statement that the square of the length of the hypotenuse of a right triangle is equal to the sum of the squares of the lengths of the two other sides, which may also be written $a^2 + b^2 = c^2$, c being the length of the hypotenuse, and a and b being the lengths of the other sides. This is because a mathematical proof of this theorem has been known since the time of Pythagoras, and because the proof of the theorem is very simple. In fact, there are many simple proofs of the theorem, so one needs to trust each proof less than if there was only a single proof.

Unfortunately, most proofs are much more complex than the proofs of the Pythagorean theorem, and often, it is not very easy to determine if a proof is correct. A famous example is Wiles' proof of Fermat's Last Theorem, which is essentially a generalization of the Pythagorean theorem. The theorem states that there are no sets of positive whole numbers a, b, c, and n that satisfy the equation $a^n + b^n = c^n$, where $n > 2$. Fermat's Last Theorem looks almost as simple as the Pythagorean theorem, but the proof of it is hundreds of pages long and took seven years to write. It seems almost inevitable that an error in the proof was found, and while the proof was eventually mended, it took more than a year [Bro15].

Certainly it would have been better if there were no errors in the proof in the first place. Unfortunately, the human mind is fallible, and often forgets or overlooks things, and this tendency only gets worse when the theorems and proofs become more complicated. Thus it seems harder to trust in the truth of Fermat's Last Theorem than in the truth of the Pythagorean theorem. It seems that what is needed is some way to make sure that the person trying to prove the theorem has remembered to take everything into account, and that the person is prevented from making errors in reasoning.

4.1 Proof assistants

A proof assistant is an interactive computer program which helps the user remember what has been proven and what has yet to be proven. It is important to note that a proof assistant is not an automated theorem prover—it cannot prove anything by itself, but is merely a tool for humans to use. Proof assistants solve the problem of forgetting a step in a proof by refusing to accept any conclusions that do not have a well-founded chain of arguments that prove the correctness of the conclusion. They do this by letting the user define theorems, functions, predicates and so on in a formal language, which the computer can then reason about to determine whether a proof is correct or not. Unfortunately, it is not possible to start the chain of arguments from nothing if the formal system is consistent—one must always take some axioms "for granted" to start the chain (as proven by Gödel, see e.g. [Smo77]). Thus even when using a proof assistant, the user is forced to trust the basic axioms of the assistant. However, one of the main reasons to use proof assistants is that it is not necessary to trust anything *but* the basic axioms of the logic of the assistant. It is important to stress that using a proof assistant does not mean that one needs to trust in any more axioms than one does when employing "manual" reasoning; proof assistants simply make the need to trust axioms explicit by disallowing implicit statements of "obvious" facts which may be accepted without further thought in less formal settings.

Since proof assistants are implemented on a computer, it is only natural that they are often able to also reason about program code, and even generate programs that implement the functions that the proven theorems are reasoning about. This means that it is possible to write a function to calculate e.g. the factorial of a number, formally prove that the function is correct, and extract a program that implements the function. All of this is only possible, of course, given that the intended functionality of the program is clear, and providing clear specifications of a program is often much harder than it may seem at first sight. Additionally, if the specification is wrong, the entire exercise becomes moot—a proof based on wrong premises will never be sound, and therefore the program will not be correct. Flaws in the specification of a complicated system may be very subtle, and thus it is important to thoroughly test any specification to make sure that it has the behaviour that is actually intended.

4.1.1 Why Coq?

Many proof assistants have been developed, but the one used in this project is Coq [Coq10]. An obvious reason for this is that CompCert is originally developed using Coq, which means that the easiest way to modify it is by using the same system. However, Coq would also have been a good choice compared to many other proof assistants if this had not been the case, for the following reasons:

- It is a very mature system, having been in development since 1984
- Its logic has dependent types (this will be elaborated on later)

- It supports extensive proof automation
- It supports code extraction/generation, which is essential to create executable verified programs.

Additionally, the "kernel" of Coq's logic (i.e. the basic axioms that underpin the entire system) is not very large, which means that it is easier to trust than otherwise comparable systems (such as F* [Swa+16]).

4.2 The theory behind Coq

Coq is an implementation of the calculus of inductive constructions and some automation features to make proofs less tedious. The calculus of inductive constructions is a type theory which can be interpreted as a typed programming language and as a constructive foundation of mathematics. To understand what these terms mean, some background is needed.

4.2.1 Intuitionistic logic

Intuitionism is the philosophical position that mathematics is solely a creation of the human mind, and that, as a consequence, mathematical truth can only be conceived by a mental construction proving the theorem in question to be true [Iem16]. Intuitionism is thus a kind of constructivism, since its followers insist that all proofs must be by construction.

Intuitionism is not compatible with classical logic since intuitionism rejects some assumptions of classical logic, and so it requires its own logic system. Logic systems that are compatible with intuitionism are called intuitionistic logics and restrict classical logic by removing the law of excluded middle and the law of double negation elimination. Thus the theorems $P \vee \neg P$ and $\neg\neg P \implies P$ can not be proven in general in intuitionistic logic.

Since intuitionistic logic differs from classical logic, it is necessary to precisely specify the meaning of the logical symbols. For an intuitionist, knowing that a statement is true is the same as having a proof of the statement. The standard interpretation of the logical symbols in intuitionism is the Brouwer-Heyting-Kolmogorov (BHK) interpretation [Tro91]. The interpretation is a structurally inductive definition of the meaning of a proof of a formula:

- A proof of $P \wedge Q$ consists of a proof of P and a proof of Q.
- A proof of $P \vee Q$ consists of a proof of P or a proof of Q.
- A proof of $P \implies Q$ is a construction (i.e. a function) which transforms any proof of P into a proof of Q.
- Absurdity (or contradiction) is denoted $\bot$ and is not provable.

- A proof of $\exists x \in A : \varphi(x)$ consists of an element of w of A and a proof of $\varphi(w)$.
- A proof of $\forall x \in A : \varphi(x)$ is a construction (i.e. a function) which transforms any proof that $x \in A$ into a proof of $\varphi(x)$.

The negation $\neg P$ of a formula P can be defined as $P \implies \bot$. It is important to note that the law of excluded middle and the law of double negation elimination are not provable in this interpretation, so the BHK interpretation is an intuitionistic logic.

While intuitionistic logic is interesting in its own right, it is not immediately obvious what it has to do with proof mechanization. For now, it suffices to say that the principle of always constructing proofs from terms will become important later.

4.2.2 The λ-calculus

The λ-calculus (pronounced and also written as "lambda calculus") is a simple formal notation for specifying functions and their application [Chu32]. It is a universal model of computation that can be used to simulate any Turing machine [Tur37], and it can thus be used as a general purpose (albeit very impractical) programming language. The main ideas of the λ-calculus are the application of a function to an argument and the formation of functions by abstraction.

The syntax of the calculus is very simple. Its alphabet consists of the left and right parenthesis, the symbol '.', the symbol 'λ', and an infinite set of variables, which are typically written $x, y, z, \ldots$. Its terms (called λ-terms) are inductively defined as follows [AK19]:

1. Every variable is a λ-term.
2. If M and N are λ-terms, then (MN) is also a λ-term (an application).
3. If M is a λ-term and x is a variable, then $(\lambda x.M)$ is a λ-term (a λ-abstraction).

When the meaning of a term is clear without a set of parenthesis, they may be omitted for readability.

An application (MN) represents calling the function M with the argument N, and could be written $M(N)$ in "regular" function notation. A lambda abstraction $(\lambda x.M)$ defines an (anonymous) function that takes an input x and substitutes it into the expression M. For example, $\lambda x.x$ is a λ-abstraction for the function $f(x) = x$.

An important point is that any lambda term, including functions, can be inputs and outputs of functions. This makes it possible to construct functions of multiple arguments by "chaining" functions such that the first function in the chain takes the first argument and returns the second function in the chain. The second function then takes the second argument and returns the third function in the chain, which takes the third argument, and so on. The process of turning a single function of multiple arguments into multiple functions of a single argument is called currying.

The actual meaning of λ-terms is defined by how the terms can be reduced [Que88]. Terms that can not be reduced are called normal forms, and when a normal form is reached, any sequence of reductions must therefore terminate.

The most important kind of reduction is called a β-reduction. It captures the principle of function application: the β-reduction of the term $(\lambda x.M)N$ is $M[x := N]$ (i.e. the term M with every free occurrence of x replaced by N).

Sometimes, attempting a function application may result in several different variables with the same name. Thus it can sometimes be necessary to rename variables before applying a function. This idea is captured in α-conversions, which allows bound variable names to be changed: the α-conversion of the term M is M with any abstraction term $\lambda x.N$ in M replaced by $\lambda y.N[x := y]$, where y can of course be any unused variable name.

It is also possible to define other kinds of reductions, but they are less important in this context.

4.2.3 Simply typed λ-calculus

In the basic (untyped) λ-calculus, there are no constraints on application, and it is even possible to apply a term to itself. This feature makes the λ-calculus very expressive, but unfortunately it also leads to inconsistency since it allows for unconstrained self-recursion [Cur41].

The simply typed λ-calculus, also denoted $\lambda^{\rightarrow}$, is a typed interpretation of λ-calculus [Chu40] [Cur34]. The types of $\lambda^{\rightarrow}$ impose constraints on the use of the application rule to avoid the inconsistency of the untyped λ-calculus. The types of a typed λ-calculus are defined by first fixing a set of base (or ground) types G, and then constructing all other types using type constructors. Additionally, a set of term constants for the base types is fixed. For simply typed λ-calculus, the exact set of base types and term constants is not very important—it is common to consider only a single base type, o, with no term constants.

In simply typed λ-calculus, there is, as the name suggests, only a single type constructor, namely the function type constructor $\rightarrow$. If A and B are both types, then so is $A \rightarrow B$.

To define the syntax of λ-terms with types, the previous definition of λ-terms is modified slightly to add type annotations to the definition of λ-abstractions, while the rest of the term definitions are the same as for the untyped λ-calculus. Additionally, any term constants are also defined to be syntactically valid λ-terms. However, simply being syntactically correct is not enough for a typed λ-term to be valid, but only enough to be considered a candidate for typed terms—a so-called pre-term. Pre-terms are thus defined inductively by:

1. Every variable is a pre-term.
2. Every term constant is a pre-term.
3. If M and N are pre-terms, then (MN) is also a pre-term (application).

4. If M is a pre-term, x is a variable, and τ is a type, then $(\lambda x : \tau.M)$ is a pre-term (abstraction).

To become a typed λ-term, a pre-term has to be well typed, i.e. it must pass a typing judgment. Typing judgments are of the form $\Gamma \vdash M : \tau$, where:

- Γ is the context (also called environment) of the judgment. The context is a set of typing assumptions used in the judgment: $\Gamma = x_1 : \tau_1, \cdots, x_n : \tau_n$. The context contains type declarations for variables that may occur freely in M.
- M is a pre-term.
- τ is the type of M.

If the typing judgment is valid then M is well typed, having type τ. A type is called inhabited if there exists at least one well typed term with the type, and uninhabited otherwise.

The validity of a typing judgment is determined using the following typing rules:

- If x has type τ in the context Γ, then $\Gamma \vdash x : \tau$.
- If c is a term constant of type τ, then $\Gamma \vdash c : \tau$.
- $\Gamma, x : \tau \vdash M : \sigma \implies \Gamma \vdash (\lambda x : \tau.M) : (\tau \to \sigma)$ (function formation).
- $\Gamma \vdash M : \tau \to \sigma \wedge \Gamma \vdash N : \tau \implies \Gamma \vdash (MN) : \sigma$ (modus ponens).

The typed λ-terms of the simply typed λ-calculus can be α-converted and β-reduced in exactly the same fashion as the λ-terms of the untyped λ-calculus, except the types of the variables must match for the resulting terms to be well typed.

4.2.3.1 λ-calculi as programming languages

Exactly as the untyped λ-calculus can be interpreted as a programming language, the simply typed λ-calculus can be interpreted as a functional programming language with types. However, the properties of the two languages are very different.

The main difference of the two languages lie in their normalization properties. A system is said to be strongly normalizing (or terminating) if, for every term, every possible sequence of reductions eventually produces a normal form. A system is said to be weakly normalizing if, for every term, there exists at least one sequence of reductions that eventually produces a normal form. It is obvious that any strongly normalizing system is also weakly normalizing.

Untyped λ-calculus is neither strongly nor weakly normalizing. To see why, consider the λ-term $(\lambda x.xx)(\lambda x.xx)$. When this term is β-reduced, each of the xs to the right of the dot in the left set of parenthesis must be replaced by the expression in the right set of parenthesis. But this produces the λ-term $(\lambda x.xx)(\lambda x.xx)$ again, so β-reduction gets stuck in an infinite loop. As this is the only possible β-reduction of

the λ-term, the system can not be weakly normalizing, and thus not strongly normalizing. This makes it possible to write programs that never terminate in the untyped λ-calculus.

In the simply typed λ-calculus, the example above is not a typed λ-term, as there is no way to construct a type that allows a function to accept itself as its argument. In fact, it can be proven that the simply typed λ-calculus is strongly normalizing [GTL03]. This means that the simply typed λ-calculus is, in fact, a total functional programming language, i.e. a language in which all possible programs terminate. While this is a very useful property, it has a major disadvantage: since it is decidable whether a program will halt or not (it always will, since all programs terminate), no total functional programming language can be Turing complete, and so there are some programs that cannot be written in the simply typed λ-calculus.

4.2.4 The Curry-Howard isomorphism

Until now it would seem that the section on intuitionistic logic and the sections on the λ-calculus are completely unrelated. But in fact, it turns out that the two concepts are very closely related.

Consider again the typing rules from the simply typed λ-calculus, but without the terms, leaving only the types:

- If a term of type τ exists in Γ, then $\Gamma \vdash \tau$ (this is the same for both variables and term constants).
- $\Gamma, \tau \vdash \sigma \implies \Gamma \vdash \tau \to \sigma$.
- $\Gamma \vdash \tau \to \sigma \wedge \Gamma \vdash \tau \implies \Gamma \vdash \sigma$.

Ignoring the Γ and $\vdash$ symbols, and interpreting the function type constructor $\to$ as implication, these rules look exactly like the rules of logic: the first rule simply says that $P \implies P$, the second rule is $P \implies Q$ and the third rule is modus ponens. In fact, the rules of simply typed λ-calculus correspond exactly to the provability rules of (implication-only) intuitionistic logic [Cur34] [How80]. This correspondence is called the Curry-Howard isomorphism, and it also extends to more complicated versions of intuitionistic logic and typed λ-calculus.

Each part of intuitionistic logic is exactly mirrored in λ-calculus, and the correspondence can be useful in both directions. The correspondence can be summarized as follows:

Intuitionistic logic		λ-calculus
Formulas	$\iff$	Types
Proofs	$\iff$	Terms
Simplification	$\iff$	Reduction
Provability	$\iff$	Inhabitation

As mentioned, simply typed λ-calculus is only expressive enough to correspond with implication-only intuitionistic logic. But in intuitionistic logic, it is also possible to prove existential and universal quantification as well as a notion of absurdity. Thus the simply typed λ-calculus must be extended to accommodate these rules of the logic.

4.3 The λ-cube

It is possible to extend the simply typed λ-calculus in several orthogonal directions. The different versions of typed λ-calculus can be classified according to the possibilities they introduce [Bar91]. This gives rise to the so-called λ-cube, which is shown in Figure 4.1. Each dimension of the cube represents a feature that introduces new possibilities in the calculus, and each arrow represents inclusion, so that e.g. $\lambda^{\rightarrow}$ is included in $\lambda 2$.

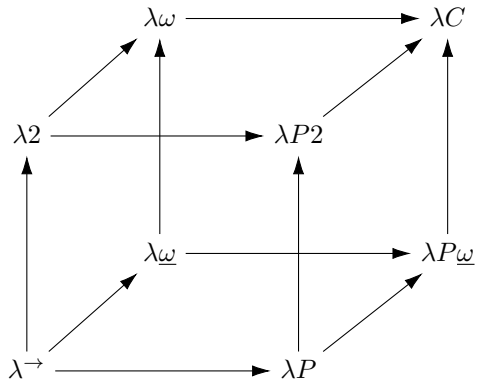

Figure 4.1: The λ-cube. Each dimension of the cube represents a feature of the calculus, and arrows represent inclusion in the sense that the calculus that is pointed to includes the calculus that is pointed from.

In the bottom left corner of the λ-cube is the simply typed λ-calculus $\lambda^{\rightarrow}$. As mentioned, the only way to construct an abstraction in this calculus is by making a term depend on a variable. The next sections will introduce several other ways of constructing abstractions.

4.3.1 $\lambda 2$

$\lambda 2$ is the polymorphic, or second order, typed λ-calculus. It is a subsystem of System F [Gir72] [Rey74]. $\lambda 2$ extends $\lambda^{\rightarrow}$ by allowing terms to depend on types. The new

abstraction is written Λ, and introduces the following typing rule:

$$\Gamma \vdash x : \tau \implies \Gamma \vdash \Lambda A \cdot x : \forall A \cdot \tau, \tag{4.1}$$

where A is not free in Γ. The symbol $\cdot$ has the same meaning as the symbol . but is raised to signify that it is used for a type variable.

Terms constructed by Λ are called polymorphic because they can be applied to different types to get different functions. The typing rule can be read as "if x has type τ, then $A \cdot x$ has type $A \cdot \tau$. This construction is very similar to what is often called generic functions in non-functional languages like Java.

It is important to note that in $\lambda 2$, the universal quantifier $\forall$ is only allowed in the context of the typing rule above. The system $\lambda 2$ corresponds to second-order propositional calculus via the Curry-Howard isomorphism [Bar91].

4.3.2 λP

λP, also called $\lambda\Pi$, extends $\lambda^{\rightarrow}$ by allowing types to depend on terms. The new abstraction is written Π, and introduces the following typing rule:

$$\Gamma, x : \tau \vdash \sigma : * \implies \Gamma \vdash \Pi x : \tau \cdot \sigma : *, \tag{4.2}$$

where the symbol $*$ represents a valid type.

The types constructed using this typing rule are often called dependent types. Dependent types are not found in many programming languages since they require a somewhat sophisticated type system to typecheck at compile time, and typechecking may become undecidable if arbitrary values are allowed in the type constructor.

The type constructor Π corresponds to the universal quantifier via the Curry-Howard isomorphism, and λP corresponds to first-order logic [Bar91].

4.3.3 $\lambda\underline{\omega}$

$\lambda\underline{\omega}$, pronounced *weak* $\lambda\omega$, extends $\lambda^{\rightarrow}$ by allowing types to depend on types. This makes it possible to construct type operators, i.e. "functions" that take a number of types as input and returns another type. The type operators themselves also have types, which are called kinds to distinguish them from "regular" types. This distinction between types and types of types (kinds) is necessary to avoid paradoxes.

More concretely, type operators make it possible to define new type constructors within the system itself. In this way, higher order constructors are formed, and $\lambda\underline{\omega}$ corresponds to the weakly higher order propositional calculus [Bar91].

4.4 The calculus of constructions

The top right corner of the λ-cube is λC, also known as the calculus of constructions (CoC) [CH88]. As shown in the cube, it includes the features of all the previously mentioned systems, so that it is possible for:

- terms to depend on terms
- terms to depend on types
- types to depend on terms
- types to depend on types.

In effect, this obscures the border between terms and types, since all types are also terms within another type themselves.

As mentioned, the Curry-Howard isomorphism links the simply typed λ-calculus to intuitionistic propositional calculus. The calculus of constructions extends the Curry-Howard isomorphism to proofs in the entire intuitionistic logic. This allows the calculus of constructions to serve as a constructive foundation for mathematics.

Every proof in the calculus is a typed λ-term. The calculus of constructions is strongly normalizing, so all computations of λ-terms terminate. While this means that the calculus of constructions is not Turing complete (as previously discussed), it has the benefit of guaranteeing that all proofs in the calculus can be mechanically verified, since they can be reduced to their normal forms.

4.4.1 Overview of the calculus

As before, the basic component of the calculus of constructions is the *term*. The expressions in the calculus are terms, and all terms have a type. Types themselves are also typed objects, and the type of a type is called a *sort*. Types and sorts are also terms, and so types and sorts can be manipulated using (many of) the same rules as any other object in the calculus.

4.4.1.1 Types, sorts, and universes

The calculus of constructions has an infinite amount of sorts, which are organized in a well-founded hierarchy. The base sorts in the hierarchy are called Prop and Set.

The sort Prop is the type of logical propositions, and objects of type Prop are called propositions. Following the Curry-Howard isomorphism, if P is a logical proposition, then it is the type of terms representing proofs of P. If there is an object p of type P, p is a witness of the fact that P is provable, i.e. p is a constructive proof of P. Conversely, if a type Q is uninhabited (i.e. no objects can have type Q), Q is an unprovable proposition.

The sort Set is the type of *small sets*. Small sets have a precise definition, but it suffices to know that they include most data types (such as booleans and integers), as well as products, subsets, and types of functions over the data types.

The rest of the infinite set of sorts consists of a hierarchy of *universe sorts* Type_i, where i is called the universe variable and can take any integer value. All the Type_i sorts contain the small sets, but they also contain *large sets* such as Set and Type_j for any $j < i$, as well as products, subsets and types of functions over these sorts.

The introduction of the universe hierarchy prevents certain paradoxes by ensuring that there is always a "bigger" sort available, so no sort has to contain itself. Since the universe variable of the Type_i sorts can be picked freely as long as the constraint that each universe can only contain smaller universes is respected, it is customary to simply write Type, so that one may write Type : Type, implicitly stating that the left sort is a smaller universe than the one on the right.

The set of sorts is called $\mathcal{S}$, and is defined as $\mathcal{S} = \{\text{Prop},\ \text{Set},\ \text{Type}_i \mid i \in \mathbb{N}\}$.

4.4.1.2 Terms

A term is constructed using the following rules:

- Set is a term.
- Type_i is a term for all integers i.
- Prop is a term.
- Variables are terms (and can have any free name such as $x, y, z, \dots$).
- Constants are terms (and can have any free name such as $c, d, e, \dots$).
- If x and y are terms, then (xy) is a term, and (xy) is read "x applied to y".
- If τ and σ are terms, and x is a variable, then $(\lambda x : \tau.\sigma)$ is a term (a λ-abstraction).
- If τ and σ are terms, and x is a variable, then $(\forall x : \tau.\sigma)$ is a term (a dependent product). If x does not occur in σ, the product is not actually dependent, and the term can be written with the simpler notation $\tau \to \sigma$.
- If τ, a, and b are terms, and x is a variable, then LET $x := a : \tau$ IN b is a term which denotes b where x is locally bound to a of type τ. This is the familiar "let–in" construct from functional programming languages such as ML.

A free variable is a variable that is not quantified or abstracted ("introduced") in a term, i.e. a variable that is not bound in the term. The notation $y\{x/t\}$ will be used to mean the term y with all free occurrences of the variable x replaced by the term t.

4.4.1.3 Typing rules

As before, terms must be well-typed to be actual terms. Whether a term is well-typed or not depends on a global environment of definitions and a local context.

The local context is a list of local variable declarations, i.e. either local assumptions $x : \tau$ or local definitions $x := t : \tau$. A local context is typically denoted Γ.

The global environment is a list of global declarations, i.e. either global assumptions $c : \tau$ or global definitions $c := t : \tau$. A global environment is typically denoted E,

and the global declarations are called constants. For both local contexts and global environments, the empty list of declarations is written $\square$ and concatenation of lists is written $\cup$.

The typing rules of the calculus of constructions define judgments of the well-typedness of terms and the validity of local contexts. The judgment $E[\Gamma] \vdash x : \tau$ means that the term x is well-typed with type τ in the global environment E and local context Γ. The judgment $\mathcal{WF}(E)[\Gamma]$ means that the global environment E is well-formed and that the local context Γ is valid in the global environment E.

The term x is judged to be well-typed in the global environment E if and only if there exists a local context Γ and a term τ such that $E[\Gamma] \vdash x : \tau$ can be derived from the following typing rules:

Empty WF $\mathcal{WF}(\square)[\square]$

Local Assumption WF $E[\Gamma] \vdash \tau : s \wedge s \in \mathcal{S} \wedge x \notin \Gamma \implies \mathcal{WF}(E)[\Gamma \cup (x : \tau)]$

Local Definition WF $E[\Gamma] \vdash t : \tau \wedge x \notin \Gamma \implies \mathcal{WF}(E)[\Gamma \cup (x := t : \tau)]$

Global Assumption WF $E[\square] \vdash \tau : s \wedge s \in \mathcal{S} \wedge c \notin E \implies \mathcal{WF}(E \cup (c : \tau))[\square]$

Global Definition WF $E[\square] \vdash t : \tau \wedge c \notin E \implies \mathcal{WF}(E \cup (c := t : \tau))[\square]$

Prop **Axiom** $\mathcal{WF}(E)[\Gamma] \implies E[\Gamma] \vdash \mathrm{Prop} : \mathrm{Type}_1$

Set **Axiom** $\mathcal{WF}(E)[\Gamma] \implies E[\Gamma] \vdash \mathrm{Set} : \mathrm{Type}_1$

Type **Axiom** $\mathcal{WF}(E)[\Gamma] \implies E[\Gamma] \vdash \mathrm{Type}_i : \mathrm{Type}_{i+1}$

Variables If $(x : \tau) \in \Gamma$ and $\mathcal{WF}(E)[\Gamma]$ then $E[\Gamma] \vdash x : \tau$

Constants If $(c : \tau) \in E$ and $\mathcal{WF}(E)[\Gamma]$ then $E[\Gamma] \vdash c : \tau$

Prop **Product** $E[\Gamma] \vdash \tau : s \wedge s \in \mathcal{S} \wedge E[\Gamma \cup (x : \tau)] \vdash \sigma : \mathrm{Prop}$
$\implies E[\Gamma] \vdash \forall x : \tau, \sigma : \mathrm{Prop}$

Set **Product** $E[\Gamma] \vdash \tau : s \wedge s \in \{\mathrm{Prop}, \mathrm{Set}\} \wedge E[\Gamma \cup (x : \tau)] \vdash \sigma : \mathrm{Set}$
$\implies E[\Gamma] \vdash \forall x : \tau, \sigma : \mathrm{Set}$

Type **Product** $E[\Gamma] \vdash \tau : \mathrm{Type}_i \wedge E[\Gamma \cup (x : \tau)] \vdash \sigma : \mathrm{Type}_i$
$\implies E[\Gamma] \vdash \forall x : \tau, \sigma : \mathrm{Type}_i$

λ-abstraction $E[\Gamma] \vdash \forall x : \tau, \sigma : s \wedge s \in \mathcal{S} \wedge E[\Gamma \cup (x : \tau)] \vdash t : \sigma$
$\implies E[\Gamma] \vdash \lambda x : \tau . t : \forall x : \tau, \sigma$

Application $E[\Gamma] \vdash t : \forall x : \sigma, \tau \wedge E[\Gamma] \vdash y : \sigma \implies E[\Gamma] \vdash (xy) : \tau\{x/u\}$

Let–In $E[\Gamma] \vdash t : \tau \wedge E[\Gamma \cup (x := t : \tau)] \vdash y : \sigma$
$\implies E[\Gamma] \vdash \textsc{let}\ x := t : \tau\ \textsc{in}\ y : \sigma\{x/t\}$

Conversion $E[\Gamma] \vdash \sigma : s \wedge s \in \mathcal{S} \wedge E[\Gamma] \vdash x : \tau \wedge E[\Gamma] \vdash \tau \preceq \sigma \implies E[\Gamma] \vdash x : \sigma$

The Conversion rule says that a term of type τ is also a term of type σ, given that the relation $\tau \preceq \sigma$ holds. The definition of the relation $\preceq$ is a bit complicated, since the relation serves multiple purposes. First of all, Prop $\preceq$ Set, Set $\preceq$ Type_1, and for any integer i, the $\text{Type}_i \preceq \text{Type}_{i+1}$, so that the relation implements the hierarchy of universes cumulatively: any term which is in a universe is also in all the larger universes, all propositions are in small sets, and all small sets are in all universes.

Like the other λ-calculi, the calculus of constructions has a notion of β-reduction. However, the calculus also has other notions of reduction, namely: δ-reduction, which expands variables into their values, and ζ-reduction, which removes local definitions in terms by replacing them by their value, as well as a concept called η-expansion, which essentially "wraps" a function term with an abstraction. The various concepts are defined as follows:

Beta $E[\Gamma] \vdash ((\lambda x : \tau.t)y) \triangleright_\beta t\{x/y\}$

Local Delta $\mathcal{WF}(E)[\Gamma] \wedge (x := y : \tau) \in \Gamma \implies E[\Gamma] \vdash x \triangleright_\delta y$

Global Delta $\mathcal{WF}(E)[\Gamma] \wedge (c := y : \tau) \in E \implies E[\Gamma] \vdash c \triangleright_\delta y$

Zeta $\mathcal{WF}(E)[\Gamma] \wedge E[\Gamma] \vdash y : \sigma \wedge E[\Gamma \cup (x := y : \sigma)] \vdash z : \tau$
$\implies E[\Gamma] \vdash \text{LET } x := y \text{ IN } z \triangleright_\zeta z\{x/y\}$

Eta Any term x of type $\forall y : \tau, \sigma$ can be identified with $\lambda y : \tau.(xy)$

The notation $E[\Gamma] \vdash x \triangleright y$ will be used to mean any one of the conversion rules above. Two terms x and y are called $\beta\delta\zeta\eta$-convertible (or just *convertible*, or equivalent) if and only if there exists terms in the global environment and local context such that $E[\Gamma] \vdash x \triangleright \cdots \triangleright t_x$ and $E[\Gamma] \vdash y \triangleright \cdots \triangleright t_y$ and either t_x and t_y are identical, or convertible up to η-expansion. A term which cannot be reduced further by conversion is called a (or said to be in) normal form as before.

The relation $\preceq$ implements the fact that terms are considered modulo conversion, so that $\tau \preceq \sigma$ if τ is convertible to σ.

4.4.1.4 Possibilities of the calculus of constructions

The definitions above essentially complete the calculus of constructions, and it is reasonably easy to see how the calculus implements e.g. universal quantification, dependent types, and polymorphism. However, some of the promised features such as a notion of absurdity and existential quantification still seem to be missing. Fortunately, it is possible to define these notions using the constructs that are already available.

Absurdity, which will be written $\bot$, can be defined by the proposition $\forall C : \text{Prop}, C$. There are no closed terms of this type, so it is impossible to construct a proof of $\bot$. Additionally, if one somehow obtains (by assumption) a term $x : (\forall C : \text{Prop}, C)$, the application of the term to a proposition, $x\ P$, is a proof of any proposition P, so that the principle of explosion holds.

Existential quantification, which will be written $\exists x : A, B$ (read: "there exists a term x of type A so that type B is inhabited"), can be defined by the term $\forall C, \mathrm{Prop}, (\forall x : A, B \to C) \to C$.

At this point, the calculus of construction can be used as a constructive foundation for mathematics, and it can also be used for higher-order functional programming and proofs about the programs one has written. However, the pure calculus of construction is not very easy to use, and it is especially hard to represent data-types efficiently.

4.4.2 The calculus of inductive constructions

The calculus of inductive constructions extends the calculus of constructions with inductive definitions. The main purpose of introducing inductive definitions is to allow efficient representations of data types [Pau15]. An inductive definition consists of a name, an arity (i.e. the type of the definition), and a set of *constructors*, each of which may take a number of parameters.

As an example, a list of elements with type A may be defined inductively as follows. The name of the inductive definition will be "list". The type of the definition will be $\mathrm{Set} \to \mathrm{Set}$. The constructors of the list will be:

- $\mathrm{nil} : \forall A : \mathrm{Set}, \mathrm{list} A$
- $\mathrm{cons} : \forall A : \mathrm{Set}, A \to \mathrm{list} A \to \mathrm{list} A$

The nil constructor should be read as: "for any small set A, nil is a list of elements of type A (the empty list)". The cons constructor should be read as: "for any small set A, an element of type A can be added to a list of elements of type A to get a new list of type A". In a typical functional programming language, these constructors might be denoted [] (the empty list) and :: (the cons operator). Using this notation, lists can be constructed using notation such as a :: b :: c :: []. It is also possible to define almost any other data type, such as the natural numbers, trees, floating point numbers, strings, and so on. Mutually inductive definitions are possible, but will not be explained in detail here.

Inductive definitions correspond more or less with the concept commonly called sum types, discriminated union types, or tagged union types in conventional functional programming languages. Inductive definitions can also be compared with the class hierarchies of object-oriented programming languages, viewing each constructor of a subclass as an inductive constructor of the parent class.

4.4.2.1 Inductive destruction

To actually use an inductive object, it must not only be possible to construct it, but also to destruct it, so that proofs by induction are possible. It turns out that there are several non-equivalent ways to define inductive destructors. In Coq, the need for consistency and strong normalization means that only primitive recursion and case analysis can be used to determine how an inductively defined object was constructed.

The current version of Coq implements inductive destructors following the ideas of Coquand [Coq92], i.e. using pattern matching and fixpoints to implement primitive recursion.

The basic idea of proof by pattern matching is that proving some property about an inductively defined object can be accomplished by proving that the property holds for each possible way to construct the object. For example, a list as defined above can either be the empty list nil, or it can be constructed from another list and an element using the cons constructor, so if a property can be proven about both of these cases, it holds for any list. This notion corresponds more or less to regular proof by structural induction (and note that since the natural numbers can be defined inductively, proof by induction is simply a special case of proof by structural induction in this system).

However, pattern matching by itself is not enough if the inductive definition is recursive, since it is then not possible to guarantee termination. Fixpoint constructions allow recursive definitions with the restriction that recursive calls to the definition must be made on terms that are structurally smaller than the original term. The precise definition of "smaller" is somewhat complicated (see [Gim95]), but it essentially means that some inductive constructor must be "unfolded" on an argument for each recursion, so that the total "chain" of constructors becomes smaller. Fixpoints allow the definition of (restricted) recursive functions and the proof of theorems on inductively defined types with recursion (which are essentially the same thing in the calculus of inductive constructions).

Thus it now becomes possible to define e.g. a function that determines the length of a list by recursively traversing the list using a fixpoint and pattern matching on each recursion to determine whether the list is the empty list or if it can be destructed further using the cons constructor "in reverse". If the list is the empty list, 0 is returned, and if the list can be destructed further, 1 plus the length of the rest of the list is returned. Note that it is allowed to define the function in this way because each recursion "removes" an element of the list under consideration, so that the "rest of the list" is always shorter than the parameter to the recursive call.

4.4.2.2 The calculus of co-inductive constructions

It is sometimes convenient to define exotic datatypes such as infinite lists, or objects with an infinite number of constructors. In the calculus of inductive constructions this is not possible, but in lazy functional programming languages such as Haskell [Mar10], infinite datatypes are extremely widespread.

To allow for (a class of) lazy data structures, Coq implements co-inductive types [Chl13]. These types allow working with some types of infinite data while preserving the property that all programs terminate. Co-inductive types differ from inductive types by allowing inductive definitions to have infinitely many constructors.

To prevent infinite loops (and thus preserve termination), co-inductive types have restrictions on recursion that are similar, but dual, to the restriction on inductive types. As such, co-fixpoints produce values of co-inductive types (while fixpoints take inductive types as arguments), with restrictions on the results of co-recursive

calls (while fixpoints have restrictions on which arguments may be passed to them recursively).

Commonly, co-inductive types can be implemented by simply removing the "base" constructor of the corresponding inductive type, e.g. the nil constructor from the list type above. This essentially imposes the condition that all objects of the co-inductive type must be infinite, since there is no constructor that does not require another object of the same type to use, and it is thus impossible to terminate the definition of a co-inductively defined object.

Since co-inductively defined objects are infinite, it is also possible to construct infinite proofs about them. This is done using co-inductive propositions, which are also useful for modelling infinite objects such as program execution traces [LG09]. In fact, a co-inductive definition of program semantics will be used to describe program evaluations that run forever, crash, or terminate in the next chapter.

4.5 An overview of the Coq syntax

The previous sections have been an overview of the calculus of (co-inductive) constructions. The notation for the calculus used in these sections can hardly be called intuitive. Luckily, the Coq proof assistant uses a less "mathematical" syntax which resembles familiar functional programming languages much more closely than what has been introduced until now. This section will explain some of the most important syntax and features of Coq through some simple examples.

4.5.1 Functions on natural numbers

One of the most familiar data structures are the natural numbers, i.e. the number $0, 1, 2, \ldots$. The natural numbers can be inductively defined in Coq using the following unary datatype:

```
Inductive natural : Type :=
| O
| S (n : natural)
```

The natural numbers are now inductively defined by the following two constructors: `O` (representing zero) is a natural number, and a natural number can be constructed by applying `S` (the successor function) to a natural number (representing the next number). Using this definition, 0 is represented by `O`, 1 is represented by `S O`, 2 is represented by `S (S O)`, and so on.

For now, all that is defined is a way of representing numbers, and so far, Coq has no idea how to interpret them. The interpretation of the defined symbols comes from definitions relating them to one another.

For example, the predecessor function can be defined using inductive pattern matching:

```
Definition predecessor (n : natural) : natural :=
  match n with
  | O ⇒ O
  | S x ⇒ x
  end.
```

This function takes a natural number `n` and returns a natural number. It works by a pattern matching on `n` (the `match ... with ... end` construct), and returning a number based on the "outer" constructor of `n`. `O` has no predecessor, but all functions in Coq must be total, so something must be returned. In this case, `O` is simply defined as the predecessor of itself to evade the problem. For all other natural numbers, `n` is "unpacked" by taking the parameter `x` of the constructor `S` and doing something with it. In this function, `x` is just returned, which has the effect of decreasing the natural number by one in the chosen representation.

In fact, it is not necessary to implement natural numbers (and many other structures) by hand, as Coq comes with a large standard library, including the natural numbers in a datatype called nat. This datatype will be used from now on instead of natural, as it implements a little syntactic sugar by allowing the programmer to write e.g. 4 instead of `S (S (S (S O)))`.

A more interesting function is the function that checks whether a number is even or not. To represent truthfulness, the following definition of boolean values will be used:

```
Inductive boolean : Type :=
| true
| false.
```

The function that checks if a natural number is even will be defined recursively, so a fixpoint is needed alongside the pattern matching:

```
Fixpoint even (n : nat) : boolean :=
  match n with
  | O ⇒ true
  | S O ⇒ false
  | S (S x) ⇒ even x
  end.
```

The function works by attempting to subtract two from the parameter `n`, then calling itself with the difference, until either the number `O` (zero) or `S O` (one) is reached. The function is allowed to call itself recursively because `x` is smaller than `n` in the sense that `x` is obtained by "unpacking" `n` twice. It is also possible to define recursive

functions of several arguments, such as the addition function:

```
Fixpoint plus (n m : nat) : nat :=
  match n with
  | O ⇒ m
  | S x ⇒ S (plus x m)
  end.
```

This recursion is allowed because the first argument becomes smaller for each recursive call.

Coq also allows the user to define new notation to make it easier to apply user-defined functions:

```
Notation "x + y" := (plus x y)
                    (at level 50, left associativity)
                    : nat_scope.
```

The newly defined notation may now be used to calculate e.g. the sum of 3 and 4:

```
Compute 3 + 4.
(* Returns
     7 : nat *)
```

4.5.2 Theorems on natural numbers

In the examples until now, Coq has merely been used as a functional programming language. But if this was the only feature of Coq, there would be no point of using it instead of a more conventional language. The main strength of Coq is the ability to define theorems and proofs of theorems. In the following, it is important to remember that proofs are functions and that theorems are the types of proofs.

A simple theorem of the natural numbers is that $\forall n, 0+n = n$, i.e. that zero is the left identity of addition. The theorem and its proof can be written in the following way in Coq:

```
Theorem plus_left_identity : forall n : nat, 0 + n = n.
Proof.
  unfold plus.
  apply (fun n : nat ⇒ eq_refl).
Qed.
```

The proof function first unfolds the definition of the `plus` function, i.e. it inserts the plus function body into the statement `0 + n = n`. Looking at the definition of `plus` above, the expression `0 + n` matches the first case in the definition, as the first argument is `0` (which is just syntactic sugar for the constructor `O`). Thus the system can apply the first case of the definition, which says that `0 + n` gives the result `n`. To prove that this is equal to the right side, the proof function next applies a λ-term (defined by the `fun` keyword) that takes a natural number and returns `eq_refl`, the proof that any object is equal to itself (reflexivity of equality). This concludes the proof, which

is terminated by the keyword `Qed`, and the proven theorem, `plus_left_identity`, can now be applied in other proofs.

Manually constructing proof terms in this way is cumbersome as soon as the proofs become complex. Instead, Coq provides so-called *tactics*, which are an alternative, more readable way to construct proofs. The proof above can be recreated using tactics as follows:

```
Theorem plus_left_identity : forall n : natural, 0 + n = n.
Proof.
  intros n. reflexivity.
Qed.
```

The first tactic introduces the variable `n` (i.e. binds the variable), and the second tactic applies the reflexivity of equality, while automatically attempting to unfold relevant definitions and simplify expressions.

Coq has many tactics such as `destruct` for inductive destructions, `rewrite` for rewriting equations using other equations, `induction` for proof by (structural) induction, `omega` for proof by the Omega decision procedure [Pug91], and `auto` and `intuition` for automatically attempting to construct proofs using the other tactics.

CHAPTER 5

Compiler verification

The calculus of co-inductive constructions is useful for proving theorems about computer programs because it is possible to reason about parts of the program as mathematical objects via the Curry-Howard correspondence. As mentioned in section 3.6, it would be useful to prove a theorem that expresses the correctness of a compiler, so that the users of the compiler do not have to worry about errors in the compiler itself. This chapter will explain the basics of how such a theorem can be expressed and proven.

CompCert [Ler09b] is an optimizing compiler with a proof of correctness. CompCert is implemented in Coq, which is itself an implementation of the calculus of co-inductive constructions, and this chapter also gives an overview of how the proof of correctness of CompCert is implemented, as well as explanations of why it is implemented in this way, while briefly discussing some of the alternative options and why they were not chosen.

5.1 Foundations

The first problem of proving the correctness of a compiler is defining what correctness means in the first place. Such a definition must capture not only the intuitive definition of the users writing programs in the source language, but also be compatible with any (formal or informal) standards and common use cases. Finally, the definition should be easy to work with from a mathematical perspective, so that the proofs can be as simple as possible.

For CompCert, the correct definition of the source language is (a large subset of) the ISO C 99 standard [99]. This standard is not a formal specification, and thus it must be translated to a specification that is compatible with formal proofs. This is an issue, as the translation from informal language to formal specification can not be proven correct. Additionally, many other compilers provide features which are not described in the standard, and so programmers may be used to features that are not supported by CompCert.

The correct definition of the target language is often described in a programmer's manual written by the processor designer or manufacturer. Unfortunately, these manuals are often written somewhat informally, and may be almost impossible to comprehend for complicated modern processor designs. This can lead to errors even

in code that has been proven "correct", as the code may have been proven to be correct according to a wrong specification. Some processor designers have begun efforts to design processors from and then distribute formal specifications of their processors to alleviate these problems, e.g. [Rei+16] [Rei16]. In this project, the target language specification is given in the Patmos Reference Handbook [Sch+19], which contains an informal description of the processor design.

Once formal definitions of the source and target languages have been determined correct, the correctness of the compiler can be expressed as a relation between programs in the source language and programs in the target language. The intuitive relation is one of preservation of semantics – that programs in the source language are compiled to programs in the target language with the same meaning as the original program. Theorems of this type can be implemented in several ways, some of which will be compared later in this chapter.

To prove the compiler correctness theorem, it is necessary to encode the meaning of the programs in the source and target languages as mathematical objects. In CompCert, this is done by parsing the source programs into abstract syntax trees which are implemented in the calculus of constructions. The abstract syntax of the programs is then assigned semantics based on the language specification.

This results in objects in the calculus of constructions that implement the source programs as functions. The compiler generates a program in the target language based on the source program, which results in another collection of objects in the calculus of constructions. The task is then to prove that these objects have the same meaning as the objects arising from the source program.

5.2 Language semantics

A programming language standard or manual will assign meaning to programs by specifying the semantics of each construct in the program. The semantics specified in the language standard is normally not a formal semantics, and so it must be translated to one that is. There are many ways to specify formal language semantics, but they can be split into three major groups: denotational semantics, axiomatic semantics, and operational semantics.

Denotational semantics, also known as mathematical semantics, give meaning to a programming language by assigning mathematical objects to each expression of the language [Sco70]. Denotational semantics thus abstract away *how* the program determines values and concerns itself only with *what* the program expressions evaluate to.

Axiomatic semantics do not base themselves on the language expressions themselves, but instead assign meaning to expressions based on a set of logical axioms [SK95, Chapter 11]. The meaning of an expression is then defined as the set of formulas that can be proved about it from the axioms.

Operational semantics do not work by attaching mathematical meaning to each expression, but instead directly construct statements about the way the program is

executed [Sco70]. Thus operational semantics are similar to interpretation, in that they assign meaning through "executing" the program. CompCert uses operational semantics, so the rest of this section will focus on the details of these.

5.2.1 Small-step and big-step semantics

There are two main categories of operational semantics: small-step semantics, which describe the individual execution steps, and big-step semantics, which describe only the overall results of the execution.

Small-step semantics, also called structural operational semantics, work by reducing the program one step at a time using a reduction relation [Plo04]. This allows small-step semantics to describe the evaluation of any program, even if the program never terminates (though this will result in an infinite reduction sequence).

In contrast, big-step semantics, also called natural semantics, work by relating programs to the final result of evaluating the programs [Kah87]. Thus big-step semantics can not distinguish between programs that never terminate (diverging programs) and programs that fail to evaluate (e.g. because they get "stuck" on an impossible command such as division by zero).

Since many programs do not terminate (in particular, embedded software is often expected to run "forever"), small-step semantics are often preferred when attempting to prove theorems on programs, especially for proving the soundness of type systems.

While small-step semantics can easily describe the evaluation of both terminating and diverging programs, it is difficult to prove correctness of even simple, non-optimizing compilation schemes using these semantics. In contrast, big-step semantics have proven easier to work with in such proofs. Fortunately, it is possible to use big-step semantics to describe diverging programs using co-inductive definitions [LG09].

In CompCert, the semantics of the target and low-level languages are small-step semantics, while the semantics of the source and high-level languages are co-inductive big-step semantics.

5.2.2 Operational semantics in CompCert

As mentioned above, several approaches to defining semantics are used in CompCert depending on the goals of the intermediate language in question. Since this project focuses mostly on the target language, i.e. the Patmos assembly language, this section will explain the approach to defining the semantics of this language in further detail.

The operational semantics are primarily defined by a transition function $T(i, S) = \lfloor S' \rfloor$ that determines the new processor state S' after executing an instruction i in initial processor state S. A processor state S is a tuple containing a register state R and a memory state M, i.e. $S = (R, M)$. The definition of T is written as a pattern matching on the instructions in the instruction set architecture. For "normal" instructions, the state is updated by incrementing the program counter and

performing any memory or register mutations needed to execute the instruction. For branching instructions, the program counter is instead set to the branch target.

For instructions internal to a function, if the program counter contains a valid pointer to an instruction i, and the registers R and memory M are compatible with the instruction, i.e. that $T(i, (R, M)) = \lfloor (R', M') \rfloor$, then the processor can transition to a new state (R', M') and begin executing the next instruction. For external calls, the semantics describe the entire invocation of the external function in a big-step style.

The transition function implements the instruction specifications using a mathematical representation of each type of value and operations over values. This representation is proven to correctly implement 32- and 64-bit arithmetic and comparisons as well as floating point operations.

5.3 Memory models

An imperative program is essentially a series of commands that modify a memory state. In fact, the modification of the memory state is typically the *only* visible consequence of executing a program.

One of the biggest challenges of verifying an optimizing C compiler is reasoning about memory accesses, since many optimization passes move, rename, and merge identifiers and memory allocations. To do this while preserving the semantics of the program, a good model of the memory is needed. C has both low-level constructs such as pointers (and even pointer arithmetic), and high-level features such as separation guarantees between memory blocks allocated by different calls to the allocation function. This makes it hard to find a balanced compromise between very abstract and very concrete memory models.

For example, it might be tempting to simply define the memory as an array of bytes indexed by machine integers. While this approach initially makes pointer arithmetic seem very simple, it also makes it impossible to enforce the separation of memory blocks allocated by different calls to the allocation function. On the other hand, a very abstract memory model can be too "strict", and make it impossible to model some features of the language, such as being able to cast between pointer types or load part of stored value as a smaller type.

5.3.1 CompCert's memory model

The memory model of CompCert takes a hybrid approach, which is able to give the expected semantics to not only ISO C99 conformant programs, but also to some industrially common classes of non-conformant programs that perform casts between pointers and exploit the memory layout of structs, unions and arrays. The memory model of CompCert is quite complex, and the following is only a short overview of how it works. For more in-depth explanations and theory the reader may consult [App+14].

In the CompCert memory model, a memory state is a collection of memory blocks. Each memory block is an array of bytes (which do not have concrete positions in the actual memory). CompCert does not model memory size, so a memory state can have an unlimited amount of memory blocks, and each block can have an unlimited amount of bytes. A pointer is represented as a tuple of a block identifier and a byte offset within the block. The block identifier is simply a positive integer.

In the C semantics, each global variable, each addressable local variable of every active function call, and each invocation of `malloc` are assigned a separate memory block. Each local variable is assigned a separated memory block which is allocated upon function entry, and deallocated when the function returns. The memory model's version of `alloc` and `free` do not model the actual C library equivalents, but simply the creation of new blocks on function entry and removal on function returns. This means that the system is free to acquire memory space through either the C library or through system calls.

An important detail is that memory blocks are non-overlapping by construction. This is a consequence of the pointer arithmetic, which can only modify the byte offset within a block, and not the block identifier itself. As an example, the addition of a pointer (b, i) and an integer n is defined

$$(b, i) + n = (b, i + n).$$

The salient point is that for two memory blocks b, b' where $b \neq b'$ it is impossible to create a pointer to b' from a pointer to b through pointer arithmetic. It is only possible to create pointers to other offsets within b, or illegal pointers.

The data type of memory states is the inductively defined type `mem`, which is inhabited by the constant `empty`: `mem`, which represents the empty memory. All other memory states are constructed from the empty memory using the following operations:

$$\text{alloc} : \text{mem} \to \mathbb{Z} \to \mathbb{Z} \to \text{mem} \times \text{block} \tag{5.1}$$

$$\text{free} : \text{mem} \to \text{block} \to \mathbb{Z} \to \mathbb{Z} \to \text{option mem} \tag{5.2}$$

$$\text{load} : \text{memory\_chunk} \to \text{mem} \to \text{block} \to \mathbb{Z} \to \text{option val} \tag{5.3}$$

$$\text{store} : \text{memory\_chunk} \to \text{mem} \to \text{block} \to \mathbb{Z} \to \text{val} \to \text{option mem} \tag{5.4}$$

$$\text{loadbytes} : \text{mem} \to \text{block} \to \mathbb{Z} \to \mathbb{Z} \to \text{option(list memval)} \tag{5.5}$$

$$\text{storebytes} : \text{mem} \to \text{block} \to \mathbb{Z} \to \text{list memval} \to \text{option mem} \tag{5.6}$$

$$\text{drop\_perm} : \text{mem} \to \text{block} \to \mathbb{Z} \to \mathbb{Z} \to \text{permission} \to \text{option mem} \tag{5.7}$$

All the operations take a parameter of type mem, which is the memory state to perform the operation on. The next paragraphs will explain the parameters and behaviour of each memory operation in more detail.

The operation alloc $m\ l\ h$ allocates a fresh memory block of $h - l$ bytes and returns a tuple of an updated memory state and an identifier for the allocated block. Valid offsets in the memory block are between l (inclusive) and h (exclusive). The memory block is not initialized, and its contents are initially undefined.

The operation free $m\ b\ l\ h$ frees (i.e. deallocates) the range of offsets from l (inclusive) to h (exclusive) in memory block b, and returns an updated memory state, or None if the address range is not freeable.

The operation load $c\ m\ b\ i$ reads a value of type c from memory block b starting at offset i, and returns the value, or None if the addresses accessed are not readable.

The relationship between the load operation and the alloc and free operations can be characterized by the following laws:

load **after** alloc**:** if alloc $m\ l\ h = (m', b)$, then

- if $b' \neq b$, then load $c'\ m'\ b'\ i' =$ load $c'\ m\ b'\ i'$
- if load $c\ m\ b\ i =$ Some v, then $v =$ Vundef.

load **after** free**:** if free $m\ b\ l\ h = m'$, then

- if $b' \neq b$ or $i' + |c'| \leq l$ or $h \leq i'$, then load $c'\ m'\ b'\ i' =$ load $c'\ m\ b'\ i'$
- if $l \leq i$ and $i + |c| \leq h$, then load $c\ m\ b\ i =$ None.

The operation store $c\ m\ b\ i\ v$ writes value v as type c to memory block b starting at offset i, and returns an updated memory state, or None if the addresses accessed are not writable.

The operation loadbytes $m\ b\ i\ n$ reads n bytes from memory block b starting at offset i, and returns a list of memvals (which will be described later), or None if the addresses accessed are not readable.

The operation storebytes $m\ b\ i\ v$ stores the bytes in the list b to memory block b starting at offset i, and returns an updated memory state, or None if the addresses accessed are not writable.

The operation drop_perm $m\ b\ l\ h\ p$ decreases the access rights (permissions) of the locations $(b, l), \ldots, (b, h-1)$ to p. The permission system of the memory model will be explained presently.

5.3.1.1 The permission system

To prevent the program from overwriting some memory locations, e.g. variables declared `const` in the original C code, the memory model has a permission system. Every byte in each memory block is either empty, or assigned one of the following permissions:

1. Freeable bytes can be compared, read from, written to, and freed
2. Writable bytes can be compared, read from, and written to
3. Readable bytes can be compared and read from
4. Nonempty bytes can be compared.

Comparability in this context means the right to compare a pointer to the byte to other pointers.

Note that the permissions have a natural order in that the each of the higher permissions include all the permissions below it. If a byte has no permission (i.e. it is empty), there are no access rights associated with it, and it cannot be referenced by a valid pointer. In fact, the memory model defines a valid pointer as a pointer (b, i) such that the byte at offset i in memory block b has a permission of at least Nonempty.

Earlier it was claimed that each byte was either empty or assigned a single permission. However, this is not completely accurate: in fact each byte is assigned both a current permission and a maximal permission. The current permission of a byte must always be less than or equal to the maximal permission of that byte. Before a byte is allocated, it has no maximal permissions, and is thus an empty byte. When a byte is first allocated, it has maximal permission Freeable. When a byte is freed, all of its maximal permissions are removed again, making it an empty byte. This ensures that it is undefined behaviour to e.g. use a pointer to a freed or unallocated byte. Once the byte has been allocated, the maximal permission can only decrease. This is the purpose of the drop_perm operation, and since there is no "lift_perm" operation, maximal permissions can only be lost, not gained. On the other hand, the current permission of a byte can both increase and decrease, provided it never exceeds the maximal permission.

The purpose of the permission system is to restrict the load and store operations (and their byte level variants), and to accomplish this, the operations check that the bytes they are trying to access have the proper permissions, i.e. at least Readable and Writable, respectively. Additionally, the free and drop_perm operations check that the affected bytes have Freeable permissions. In all cases, the permissions that are checked are the current permissions.

The load and store operations (and their byte level variants) do not change the permissions of the bytes they access, but the other operations do. The operation alloc $m\ l\ h = (m', b)$ sets the maximal and current permissions of the bytes $(b, l), \ldots, (b, h-1)$ to Freeable. The operation free $m\ b\ l\ h$ removes all permissions from the bytes $(b, l), \ldots, (b, h-1)$, thus making them empty bytes. The operation drop_perm $m\ b\ l\ h\ p$ sets the maximal and current permissions of the bytes $(b, l), \ldots, (b, h-1)$ to p.

5.3.1.2 Data representations

In existing CompCert backends, a memory value can be a 32-bit machine integer `Vint(i)`, a 64-bit machine integer `Vlong(i)`, a 64-bit double precision float `Vfloat(f)`, a 32-bit single precision float `Vsingle(f)`, a pointer `Vptr(b,i)`, or an unknown (undefined) value `Vundef`. A value is annotated by a "memory chunk", which indicates the type, size, and signedness of the value. The memory chunk of a value defines how many and which type of bytes it can be decomposed into or composed from. The following memory chunks are defined:

- `Mint8signed`, for signed 8-bit integers
- `Mint8unsigned`, for unsigned 8-bit integers
- `Mint16signed`, for signed 16-bit integers
- `Mint16unsigned`, for unsigned 16-bit integers
- `Mint32`, for 32-bit integers or pointers
- `Mint64`, for 64-bit integers
- `Mfloat32`, for 32-bit single precision floats
- `Mfloat64`, for 64-bit double precision floats
- `Many32`, for any 32-bit value
- `Many64`, for any value.

Each memory chunk c comes with a size $|c|$ in bytes and an alignment $\langle c \rangle$.

In Equation 5.4, the store operation is defined as returning a value of type option mem. This is not only because the target address needs to be writable, but also because the operation performs alignment checking: the store can only succeed if $\langle c \rangle$ divides the byte offset of the store operation. The load operation in Equation 5.3 also performs alignment checking in a similar manner.

It is also possible to work with individual bytes in the memory model. A memory state includes a mapping which associates a memval to each byte (b, i). The memval determines which of the following three categories the byte belongs to:

- Undef represents bytes which have undetermined bit patterns
- Byte n represents the actual 8-bit integer $n \in [0, 255]$
- Pointer b i n represents the nth byte of a pointer to (b, i).

The idea is that integer and float values are decomposed and stored in the memory as a sequence of bytes, taking the hardware endianness and encoding into account. On the other hand, the hardware representation of pointer values is hidden from the memory model.

The composition and decomposition of bytes to and from memory values is handled by an encode (respectively decode) function, which takes a memory chunk and a value (respectively a memory chunk and a list of bytes) and returns either a list of memvals, which must all be either Bytes or Pointers, depending on the original value (respectively a value depending on the memory chunk). When encoding a value, the value may be normalized to fit in the specified memory chunk by removing the high bytes of the value. The decode function is carefully selected to be the left inverse of the encode function, except for the possible normalization. This means that encoding, then decoding a value will yield the original value, provided appropriate memory

chunks are picked so no normalization takes place. This also allows the memory model to access the byte-level representation of floating point numbers by encoding them as bytes, then decoding them into integers of an appropriate size. Note that this makes it possible to "generate" and load integer and float values that have never been stored in the memory by loading byte-level representations of other integers or floats.

The encoding and decoding functions can fail if it is not possible to fit the bytes in the specified memory chunks. In this case, the encoding function will return a list of Undef memvals, and the decoding function will return an undefined value Vundef.

The properties of the memory operations are characterized by the following laws.

loadbytes **after** storebytes**:** if storebytes $m\ b\ i\ B = \text{Some}\ m'$,

(compatible types) loadbytes $m'\ b\ i\ |B| = \text{Some}\ B$

(disjoint) if $b' \neq b$ or $i' + n' \leq i$ or $i + |B| \leq i'$, then loadbytes $m'\ b'\ i'\ n' =$ loadbytes $m\ b'\ i'\ n'$.

load **after** store**:** if store $m\ c\ b\ i\ v = \text{Some}\ m'$, then

(disjoint) if $b' \neq b$ or $i' + |c'| \leq i$ or $i + |c| \leq i'$, then load $m'\ c'\ b'\ i' =$ load $m\ c'\ b'\ i'$

(compatible types) if $|c'| = |c|$, then load $m'\ c'\ b\ i = \text{Some}(\text{convert}\ c'\ v)$

(incompatible types) if $|c'| \neq |c|$ and load $m'\ c'\ b\ i = \text{Some}\ v'$, then $v' = \text{Vundef}$

(overlapping) if $i' \neq i$ and $i' + |c'| > i$ and $i + |c| > i'$, and load $m'\ c'\ b\ i' = \text{Some}\ v'$, then $v = \text{undef}$.

Pointer value integrity As mentioned, it is possible to load integer and float values that were never stored, but an important property of the memory model is that it is not possible to load pointers that were never stored. More precisely, if a pointer value is loaded after a store, it is either the case that the pointer value is the stored value, or that the load is disjoint from the store, so that the loaded pointer value was already present in the memory.

This integrity is important for the proofs of invariance by memory transformation in CompCert, which only hold if the memory model can guarantee that pointer values can not be generated "out of thin air".

5.3.1.3 Memory transformation

As mentioned above, each memory block in the CompCert memory model is identified by a positive integer, the block identifier. The block identifiers are assigned sequentially at each allocation. An important capability of the CompCert memory model (and the formal semantics of CompCert C) however, is that the precise choice

of block identifiers do not matter. That is, both the formal semantics of CompCert C and the memory model are invariant to renaming of block identifiers.

Some passes of the CompCert compiler need stronger properties than simply renaming invariance for their correctness proofs. This is the case whenever the compiler needs to pack together memory blocks (e.g. to create a stack frame) and whenever it needs to extend an already existing memory block (e.g. to spill local variables).

CompCert introduces two function types to solve these problems: memory injections, which is memory block renamings allowing blocks to be "put into" other memory blocks as sub-blocks, and memory extensions, which allow blocks to be extended with more fields. A memory injection I is a function with type block $\to$ option(block $\times$ $\mathbb{Z}$). It is defined as follows. Let b be a block identifier. Then $I(b) = \text{None}$ means that the memory block identified by b was removed from the memory by a program transformation. If $I(b) = \text{Some}(b', n)$, then the memory block identified by b was put into block b' as a sub-block starting at offset n.

A memory injection I induces a relation $I \vdash v_1 \hookrightarrow v_2$ between the values v_1 of the original program and the values v_2 of the transformed program. The relation can be characterised by the following inference rules:

$$\frac{}{I \vdash \text{Vundef} \hookrightarrow v_2} \qquad \frac{}{I \vdash \text{Vint}(i) \hookrightarrow \text{Vint}(i)}$$

$$\frac{}{I \vdash \text{Vfloat}(f) \hookrightarrow \text{Vfloat}(f)} \qquad \frac{I(b_1) = \text{Some}(b_2, n) \quad i_2 = i_1 + n}{I \vdash \text{Vptr}(b_1, i_1) \hookrightarrow \text{Vptr}(b_2, i_2)}$$

The top left inference rule allows undefined values in the original program to be replaced by defined values by a program transformation. The top right and bottom left rules specify that integers and floating point numbers are simply left untouched by memory injections, as their values do not depend on their position in memory. The bottom right inference rule specifies how to inject memory block b_1 into block b_2 using I, i.e. by adding the block offset to the original offset within b_1 and using the result as the offset in b_2.

A memory injection I also induces a relation $I \vdash m_1 \mapsto m_2$ between the memory state m_1 of the original program and the memory state m_2 of the transformed program. This memory relation is defined as follows:

$$\begin{aligned} & I(b_1) = \text{Some}(b_2, n) \wedge \text{load } c\ m_1\ b_1\ i = \text{Some}(v_1) \\ \implies & \exists v_2, \text{load } c\ m_2\ b_2\ (c + n) = \text{Some}(v_2) \wedge I \vdash v_1 \hookrightarrow v_2 \end{aligned}$$

Memory injections have nice properties of commutation with the operations of the memory model [LB08], which supports the proofs of semantic preservation for those optimization passes that modify the memory layout or operations of the program by allowing proofs to "move" memory operations and values between the original and transformed programs.

5.4 How to prove preservation of semantics

As mentioned above, the main goal of formally defining the semantics of the languages is to be able to prove that the compiler preserves the semantics of the source program S in the compiled program C in the target language.

The notation $S \Downarrow B$ will be used to denote that the program S executes with the observable behaviour B. Programs in CompCert can be divided in three categories depending on their observable behaviours: terminating programs, diverging programs, and programs that "go wrong" e.g. by dividing by zero or accessing an array out-of-bounds. Behaviours also include a "trace" of the input and output operations that were performed during the execution of the program. This trace is constructed using the formal semantics of the language, the memory model, and the formal model of the register files.

An obvious definition of a compiler that preserves semantics is one that respects the following relation:

$$\forall B, S \Downarrow B \iff C \Downarrow B \ , \qquad \text{(Bisimulation)}$$

i.e. that S and C have exactly the same observable behaviours.

Unfortunately, this definition is too strong to be usable for real languages. Since many source languages are not deterministic (e.g. the evaluation order of expressions in C is non-deterministic), but most processors are deterministic, the compiler must be able to select one of the possible behaviours of the source program when generating the target program. It may also be the case that the compiler optimizes away a "going wrong" behaviour such as a division by zero assigned to a variable that is never used.

To account for these problems, the relation may changed to

$$S \text{ safe} \implies (\forall B, C \Downarrow B \implies S \Downarrow B) \ , \qquad \text{(Backward simulation)}$$

where S safe means that S has no "going wrong" behaviours. This relation guarantees that if S does not go wrong, C will also not go wrong, and that any behaviour of C will be one of the allowed behaviours of S.

An alternative relation is

$$\forall B : B \notin \text{Wrong}, \ S \Downarrow B \implies C \Downarrow B \ , \qquad \text{(Forward simulation)}$$

i.e. that all possible behaviours of the source program are also behaviours of the compiled program, provided that the behaviours are not wrong.

Forward simulations are generally easier to prove than backward simulations [Ler09a], but they are not as strong: even if all behaviours of the source program are also behaviours of the compiled program, the compiled program could also have additional, unwanted behaviours. Fortunately, if the target program is deterministic, it allows only one behaviour, and thus can have no additional behaviours. The target program is deterministic if the target language and the execution environment are deterministic.

This "trick" makes it possible to prove semantic preservation using the easier forward simulation approach, as forward simulation thus implies backward simulation

provided that the compiled program is deterministic. In CompCert, the strategy is therefore to prove forward simulation for safe programs, prove that the target language is deterministic, and the combine these proofs to yield the final proof of semantic preservation.

5.5 The CompCert proof structure

The previous sections have explained the individual ideas that go into the proof of preservation of semantics for CompCert. This section will give an overview of the overall structure of the proof, and how some of the individual components fit together.

As mentioned, a program is characterised by its observable behaviour, i.e. the way it affects the register files and memory (and any memory-mapped peripherals). The behaviour of a program is represented by a trace of the input and output operations performed during the program execution.

The overall theorem of correctness consists of two parts: for programs that do not "go wrong", the compiled program must only have behaviours that are also behaviours of the source program (forward simulation of safe programs), and the compiled program must be deterministic. The proof of determinism of the compiled program is comparatively easy, as a single threaded assembly language is not very complex.

The proof of forward simulation of safe programs is significantly more complex, as it also involves proofs that all optimizations are correct. Modeling the compiler as a total function Compile returning either an Error or a compiled program $\mathrm{OK}(C)$, the first part of the theorem can be precisely stated:

$$\forall S, C, B \notin \text{ Wrong}, \ \mathrm{Compile}(S) = \mathrm{OK}(C) \wedge S \Downarrow B \implies C \Downarrow B.$$

The compiler is structured as a sequence of compilation passes. As each pass is independent of the others, the passes can also be proved correct independently. CompCert differs from many other compilers by using numerous intermediate languages to implement these passes. Since the proofs of each pass need to be composed together to create a proof of the overall theorem, each intermediate language must also have formal semantics.

Each pass can either be proven directly, or the pass may be implemented using non-trusted code, the results of which are then verified a posteriori using a formally verified validator.

Many parts of the compiler are very similar. For example, the syntax of source, intermediate, and target languages all share a common shape. The code of CompCert is structured to take advantage of this fact by defining an infrastructure of proof modules that can be reused. This infrastructure includes definitions and theorems on abstract syntax trees, global environments, the memory model, small-step semantics, and values. These modules are used for defining each of the languages used in the compiler passes of CompCert, and for proving properties about them.

CHAPTER 6

Porting CompCert to Patmos

The new work in this thesis is a port of the CompCert compiler to the Patmos processor. This combines the formally certified correctness of CompCert with the time-predictability of Patmos to enable the creation of systems that are both functionally correct and fast enough to be safe. These two properties combined enable a new level of confidence in embedded systems by providing a platform on which formally verified programs can be safely executed and analysed for their timing behaviour.

6.1 Overview

The CompCert development consists of a number of program and proof modules that are composed to create the overall program and (simultaneously), the proof of correctness. The development is split into several parts, notably a set of common modules, a set of frontend modules, a set of backend modules, and a set of architecture specific modules for each of the supported instruction set architectures.

To port CompCert to the Patmos architecture, a new set of modules were written to model the formal semantics of the Patmos assembly language and couple this model to the existing intermediate languages of CompCert.

6.2 Methodology

CompCert is an existing compiler with several backends for different architectures.

The Patmos port was based mostly on the RISC-V backend, as this architecture is the most similar to the Patmos architecture, but for some modules, inspiration was also found in the other backends, particularly for the ARM and PowerPC architectures.

The modules containing the program definitions and proofs of CompCert depend on each other to form a directed acyclic graph of dependencies. Following the dependency graph, the existing backend was modified module by module to model Patmos instead of RISC-V. An overview of the work done can be found in Figure 6.1, which

displays all modified modules as well as those modules that will have to be modified to obtain a fully functional backend for Patmos. The green modules in the figure are those which are currently functional, while the yellow modules are not. The white modules are those on which no work has been done. Each of the modules to the extreme right are used by other parts of the compiler to implement optimizations and code generation.

6.3 Modifying the backend infrastructure

Unfortunately, it was necessary to modify not only the designated architecture specific modules, but also some common and backend modules, since the existing infrastructure of CompCert makes some assumptions about processor capabilities that do not hold for Patmos. This was an initially unforeseen workload which slowed down the development of the Patmos backend significantly.

The main difference between Patmos and the architectures already supported by CompCert is that Patmos does not have 64-bit registers to store double precision floating point numbers and 64-bit integers. This means that the infrastructure modules had to be changed to accommodate splitting 64-bit values into two 32-bit registers. This was only done for integer values, while it was decided to simply not support 64-bit floating point values to keep the model as simple as possible. The C99 standard allows the double data type to have any size at least as large as the float (single precision) data type, so keeping both data types as 32-bit values is standard compliant [99].

The main modification to the infrastructure was to the formal model of the type system of the languages, as the double type had to be changed to a 32-bit type. This in turn required modifications to the models of initial values of variables and memory locations, as well as to the load and store operations and the definition of memory chunks in the memory model.

Unfortunately, these modifications broke several proofs and definitions, which had to be amended to accommodate the changes. Not all the proofs and definitions were successfully restored; specifically, the definitions of the functions to extract the values of the arguments of external calls using the calling conventions are not currently valid. Since these definitions are missing, it was not possible to define the functions that simulate execution of instructions and whole programs in the formal model of the Patmos ISA.

It would also have been possible to modify the frontend of the compiler to directly implement that float types are 32-bit on Patmos, but this would still have required the removal of 64-bit floating point code in the backend, and would most likely have broken even more proofs and definitions than the chosen approach.

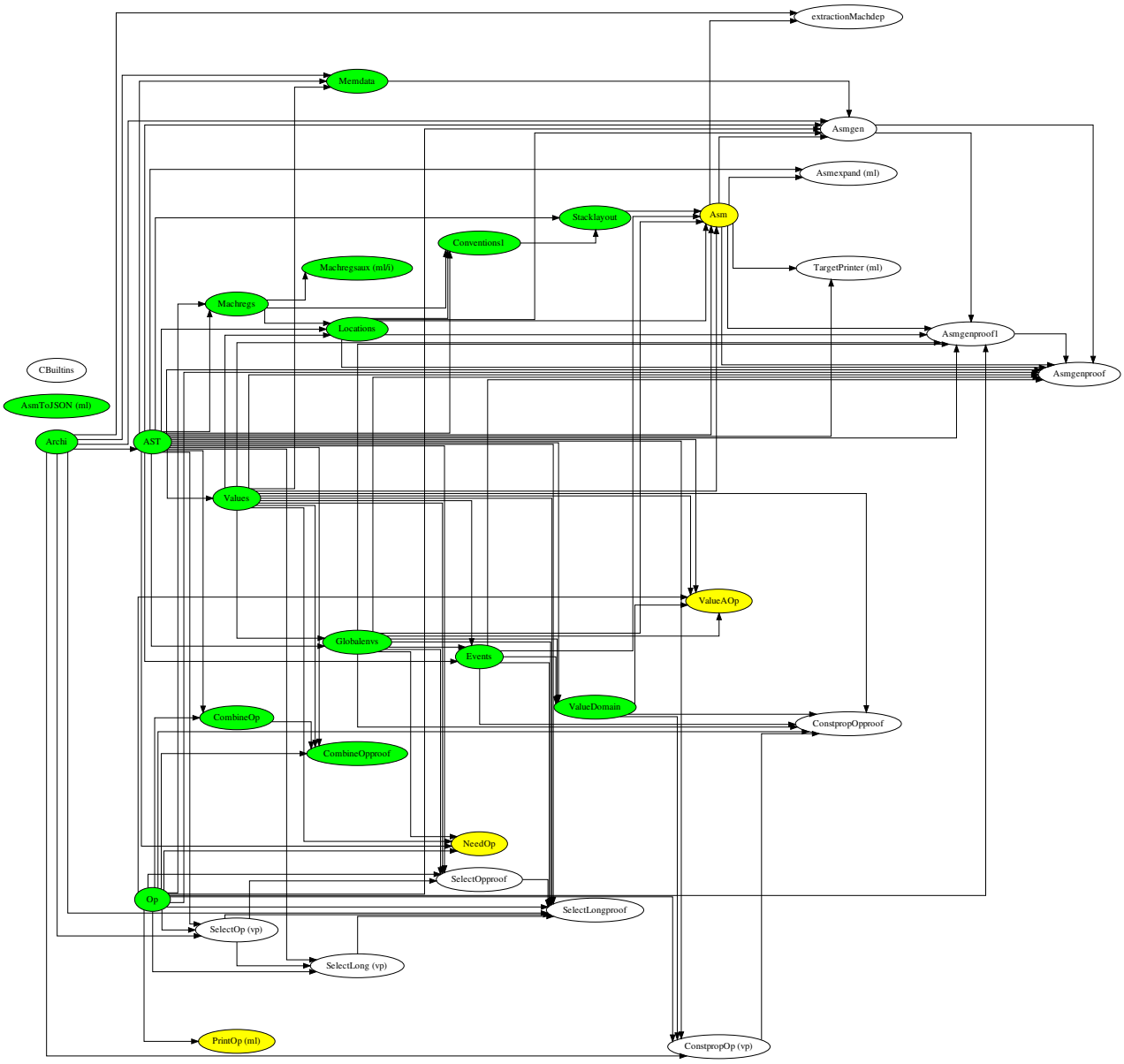

Figure 6.1: Graph of dependencies of the modified and new CompCert modules. Green modules are functional, yellow modules are semi-functional, and white modules have not yet been modified. The modules are ordered by dependency from left to right such that modules to the extreme left depend on no other modified modules, while modules to the extreme right depend on many other modules.

6.4 A formal model of the Patmos ISA

To generate correct assembly code for the Patmos processor, a formal model of the processor must be implemented. The model is used to prove that the generated assembly instructions for the processor has the same semantics as the statements in the source program. Thus the formal model of the Patmos processor must correspond exactly to the actual behaviour of the processor implementation that the generated code is to be executed by. Unfortunately, an exact formal model of a processor is in most cases too cumbersome to work with, but it is in many cases possible to avoid modelling all behaviour of the processor by ignoring certain features. This will not result in an invalid correctness proof as long as care is taken not to generate instructions that depend on the features that were not modelled. The CompCert backend for Patmos ignores or simplifies several features, which are detailed in the following sections.

6.4.1 Pipelines

The Patmos processor features several pipelines: an instruction pipeline and a separate multiplication pipeline. The pipelines of the processor are not modelled explicitly. Instead, non-delayed versions of all branch instructions are generated by the assembly generator, and no-operation instructions are inserted after every multiplication instruction to "wait out" the parallel multiplication pipeline. While this approach is not very efficient, it simplifies the formal model since it avoids the need to model the pipelines.

6.4.2 VLIW features

The CompCert compiler does not currently support any architectures with very large instruction words, and it is expected that the CompCert development will require significant changes to model dual issue features. For this reason, the dual issue feature of Patmos is ignored completely in the port, and the compiler simply generates code for only one of the two instruction slots. This is no problem for the Patmos processor, but the generated assembly code will obviously be less efficient (in most cases) than if both instruction slots were used.

6.4.3 Memories

Patmos has several memories, but only the global memory is used in the compiler to simplify the model, i.e. only load and store instructions with the "m" suffix are modelled.

The load instructions incur a latency of a single cycle before the loaded value can be used, so the compiler inserts a single no-operation instruction after each load

instruction to wait for the value to become available. This avoids the need to formally model the pipeline to determine when values are available.

6.4.4 Registers

Patmos has the following registers:

- 32, 32-bit general-purpose registers: r0–r31
- 8, single-bit predicate registers: p0–p7
- 16, 32-bit special-purpose registers: s0–s15.

The pipeline implements full forwarding, so the result of register writes is available immediately.

r0, p0, and s0 are treated specially: r0 is always 0, p0 is always 1, and s0 can not be written to except for the first 8 bits, which are an alias for p0–p7. The three different register types are each modelled using two inductive types: one type for all registers except the first and one type for all registers. This makes it possible to remove the possibility of using read-only registers in "destination slots" in the definition of the instruction syntax. There is also an inductive type for all registers, which has a constructor for each of the three register types plus a constructor for the program counter.

Conventional names for some dedicated registers are defined:

- r30, the frame pointer FP
- r31, the stack pointer SP
- s2, the low multiplication result SL
- s3, the high multiplication result SH
- s5, the spill pointer SS
- s6, the stack pointer ST
- s7, the return base address SRB
- s8, the return address offset SRO
- s9, the exception return base address SXB
- s10, the exception return address offset SXO.

6.4.5 Instruction set

The instruction set of the Patmos architecture is modelled by an inductive type of instructions and a function that defines the semantics of each instruction by describing how the memory and registers are affected by each instruction (including the program counter to model branches). The inductive type is called `instruction` in the implementation, while the semantic function is called `exec_instr`. The following sections will describe the choices taken in implementing the formal model.

6.4.5.1 Abstract syntax

In the following, a label is simply a positive integer, while a predicate is either a predicate register or an inverted predicate register. The instruction type is inductively defined with a constructor for each addressing mode. For example, the register–register integer addition instruction is defined as

```
Padd (p: predicate) (rd: ireg) (rs1 rs2: ireg0)
```

Notice that `rd` can not be r0, while `rs1` and `rs2` can. The rest of the arithmetic instructions are more or less identical to the addition instruction.

The 12-bit immediate arithmetic instructions are also defined similarly, except that `rs2` is replaced by an immediate integer argument, e.g.

```
Psubi (p: predicate) (rd: ireg) (rs1: ireg0) (imm: int)
```

Note also that the instruction names are post-fixed with i to indicate that they are immediate.

The definitions of the 32-bit immediate (long) arithmetic instructions diverge somewhat from the Patmos assembly specification, as they are treated as single instructions instead of a bundle consisting of an instruction and a 32-bit value. The reason for doing this is that treating the instructions like all the others makes the definition of instructions much simpler and more uniform, at least as long as instruction bundling is not modelled. This means that the long immediate instructions are simply of the same format as the 12-bit immediate instructions.

The modification does not have a great impact on the output, since the Patmos assembly generator can simply expand the instruction to the proper format when generating the actual assembly code. Since the long immediate instructions fill a whole bundle by themselves, it is enough to forbid them from being bundled with other instructions to avoid problems if the dual-issue features of Patmos are eventually modelled.

It should be noted that the Patmos specification sets various restrictions on the immediate operands to the instructions (primarily length of the immediate). These restrictions are not captured in the abstract syntax or in the semantics, but are deferred to the assembly generator.

All the other instructions essentially follow the same pattern of requiring a predicate and one or more operands from either the general registers, special registers,

predicate registers, or immediate values. The return functions are special in that they only require a predicate.

Besides the actual instructions of the Patmos instruction set, some pseudo-instructions are also defined. These are:

- allocframe, to allocate a new stack frame (which can not be expressed fully in the memory model)
- freeframe, to free a stack frame
- label, to define a code label
- btbl, to create an N-way branch through a jump table
- builtin, to call built-in functions
- nop, to do nothing.

The code of an assembly program is defined simply as a list of instructions.

6.4.5.2 Operational semantics

In the semantics, the register files are modelled as a simple mapping from registers to values. The model maintains that integer registers are mapped to values of type int and boolean registers to either type zero or type one. Additionally, the model maintains that r0 always equals 0, that p0 always equals 1, and that s0 is an alias for p0–p7 as previously mentioned.

The semantics are purely small-step, and are a function from the current state to either an updated state, or the special outcome `Stuck` in case the processor is stuck. A state is defined as a pair consisting of a register set and a memory state.

The semantics of the arithmetic instructions are defined using a CompCert module that implements machine integer operations. Most of the arithmetic instructions are very straightforward, e.g. adding two registers and storing the result in a third register.

Shift instructions only use the last 5-bits of the argument for the shift amount. This is implemented in basic machine integer operations by shifting the argument 27 places left, then 27 places right to trim the value to 5 bits.

Patmos does not have a dedicated move instruction, but the operation can be implemented by adding zero to the source register and storing the result in the destination register, or by adding an immediate value to r0 (which is always zero) for immediate moves.

Move instructions to and from special registers simply copy the value between registers, except the move-to-special instruction for special register s0, which instead copies the first 8 bits of the source register to the predicate registers (except p0, which always has the value one).

To simplify the semantics, only the load and store instructions for global memory are modelled, and stack control instructions are not modelled at all. The load and

store instruction semantics use the memory model load and store functions almost directly to represent modifying the memory. All global memory load and store instructions are modelled, so it is possible to load and store words, half-words, and single bytes.

The memory offset ("dis" or displacement in the implementation) to load or store from is a 7-bit unsigned integer, but the semantics do not model this, instead relying on the code generation module to respect this limitation.

Most instructions add one to the program counter to advance the processor. This is implemented by the `nextinstr` function, which simply offsets the pointer stored in the program counter by one.

Call instructions instead go to a label by setting the program counter to a pointer representation of the label. If it is not possible to find the label position in the program, the processor becomes `Stuck`. The call instruction also stores the current program counter in the SRB and SRO registers. If this is not possible, the processor also becomes `Stuck`.

Local branch instructions go to a relative offset from the current program counter by simply offsetting the pointer stored in the program register. If this is not possible, the processor becomes `Stuck`.

Global branch instructions ("branch with cache fill") go to a label in the same way as a call instruction, but does not store the current program counter.

Return instructions returns to the previous call instruction by setting the program counter to the sum of the SRB and SRO registers.

Only "non-delayed" version of each control flow instruction are modelled in the current implementation.

All instructions are fully predicated, but this is currently only modelled for some instructions. To model the predication of an instruction, the function simply checks whether the predicate is true or not, inverts the result if necessary, and immediately increments the program counter if the instruction is not to be executed. If the instruction is to be executed, the function models the actual "content" of the function, and only then increments the program counter.

6.4.6 Application binary interface

The application binary interface (ABI) of a program specifies a convention of how program modules can communicate with each other. The application binary interface of the CompCert compiler backend for Patmos is modelled after the ABI of the LLVM compiler for Patmos.

Arguments to a function are stored in registers when possible. Up to six registers are used for this purpose, namely registers r3 through r8. For 64-bit arguments, the value is stored in two registers: the high part of the value is stored first, then the low part. If it is not possible to store an argument in a register, it is instead stored on the shadow stack via the global memory.

Register r0 is defined to be zero at all times, and this is actually also implemented in hardware.

Registers r1 and r2 are used to return values from function calls. For 64-bit arguments, the value is stored in two registers: the high part of the value is stored in register r1, and the low part is stored in register r2.

Register r29 is used as a temporary register. Register r30 is used as the frame pointer, and register r31 is defined as the stack pointer for the shadow stack in global memory.

Registers r1 through r19 are caller-saved registers, while registers r20 through r31 are callee-saved registers.

All predicate registers are caller-saved registers.

Special register s0, which represents the predicate registers, is a callee-saved register. The special registers SS (s5) and ST (s6) are callee-saved registers. The special registers SRB, SRO, SXB, and SXO (registers s7-s10) are callee-saved registers. All other special registers are caller-saved registers.

All stack data is in the global memory, and the stack grows from top to bottom.

6.5 Architecture-dependent optimizations

Several of the optimizations implemented in CompCert depend on the model of the instruction set architecture for their correctness proofs. These optimizations include combined operation recognition for common subexpression elimination, value analysis for static evaluation, and neededness analysis for dataflow optimizations.

No new, Patmos-specific optimizations were added to any of the modules implementing the architecture-dependent parts of these optimizations. Instead, the modules were only modified by removing optimizations that are not possible to implement for the Patmos processor, e.g. optimization of 64-bit operations.

6.6 Coupling the Mach language to Patmos assembly

The link between the formal model of the Patmos architecture and assembly language and the rest of CompCert is the Mach language, which is the final intermediate language after all optimization passes have been performed.

The Mach language is a sort of abstracted assembly language, but with actual memory locations in the stack frames and a finite amount of actual registers, which match the registers of the processor. The Mach language for Patmos has access to registers r1 to r28, as the remaining registers are reserved as follows: r0 is always zero, r29 is reserved as a temporary register for the assembly-code generator, r30 is the frame pointer, and r31 is the stack pointer. Mach does not have access to special registers or predicate registers, but functions using only “normal” registers and a wider selection of instructions that can be translated to instructions that use these Patmos-specific features.

Since the definition of the formal model of the Patmos assembly language was not fully completed, it was not possible to define the functions that actually translate Mach code into Patmos code, but this would be the next logical step after defining the formal model of the Patmos language.

The Patmos assembly language does not contain any 64-bit or floating point instructions, but the Mach language does. Thus any 64-bit Mach instructions must be expanded to several Patmos instructions, while floating point Mach instructions must be expanded to function calls to a soft-float library in the runtime.

6.7 From Patmos assembly to machine code

The CompCert compiler itself does not generate machine code, but only code in the assembly language of the target architecture. Thus the compiler needs an external assembler to actually generate executables. CompCert uses the GNU Assembler to assemble the generated assembly code into machine code. Since the only available (fully functional) assembler for Patmos is the one included in the LLVM compiler, CompCert would have to be modified to use the LLVM assembler to actually generate executables for Patmos.

Additionally, Patmos code needs a runtime including a soft-float library to work with floating point numbers of any kind. Such a runtime could most likely be constructed based on existing code from e.g. the LLVM compiler for Patmos, but it is likely that the runtime would need to be modified to be compatible with assembly code generated by CompCert.

CHAPTER 7

Results

The goal of this project was to implement a CompCert backend for the Patmos processor. Unfortunately, this goal was not completed, as only parts of the backend were implemented.

Since the compiler backend was not completely implemented, it has not been possible to test the performance of the compiler or compare it to the existing compiler toolchain based on LLVM. It is expected that the CompCert port, if finished according to the original goal of the project, will generate slower assembly code than the existing compiler, as it would not implement any Patmos-specific optimizations.

7.1 CompCert C and the C standard

CompCert does not support the entire C language, but it does support a major subset of ISO C 99.

The only language features that are not supported are as follows:

- Switch statements must be structured, so unstructured constructions such as "Duff's device" are not supported.
- The longjmp and setjmp statements may or may not work.
- Variable-length array types are not supported.

Fortunately, this is not a very obtrusive limitation of the language, as these features are rarely needed, or even allowed, in critical systems programming.

7.2 Implemented optimizations

Since the Patmos backend was not finished during the project period, no optimizations were fully implemented. However, the infrastructure for several optimizations is available in the implemented parts of the backend.

7.2.1 Patmos-specific features

Since the dual-issue pipeline of Patmos was not modelled in the current implementation, it is not possible for the backend to generate instructions to be executed in

parallel. This is a major limitation on the performance that can be expected from code generated by the compiler.

Additionally, the compiler does not currently model the delay slots of instructions, and thus does not attempt to fill the delay slots, simply inserting no-operations instead. This means that, at a minimum, several cycles are "wasted" after every multiplication, branch, function call, and return instruction. It should be possible to implement a data flow analysis to let the compiler attempt to fill delay slots with instructions using non-interfering registers. For control flow instructions, it may even be possible to simply rearrange the emitted instructions so the "end" of the instruction block is moved the appropriate number of slots after the control flow instruction.

CHAPTER 8

Related work

While CompCert itself is in active development, there are also currently several "spin-off" projects attempting to interface with the compiler or implement new features.

The Patmos processor is also in active development.

8.1 Verified optimizations for VLIW processors

CompCert does not currently attempt to reorder operations for performance optimization. This does not make a very large difference for out-of-order processors or processors with speculative execution, since the hardware will reorder operations on its own.

On VLIW processors such as Patmos, however, the assembly code is expected to explicitly state which instructions should be executed in parallel. This project does not attempt to modify CompCert to schedule instructions, but simply outputs instructions nearly in the order they appear in the source code, and in only one of the two instruction slots of Patmos.

The Proofs and Code analysis for Safety and Security team at Verimag has developed an extension of CompCert for VLIW processors that allows the compiler to optimize VLIW assembly by scheduling instruction bundles within basic blocks [SBM20]. The scheduler consists of an unverified oracle that does the actual scheduling (with various implementations), and a certified validator that proves semantic preservation. The developed extension is independent of the specific instruction set architecture and is also largely independent of the rest of CompCert's passes, so it is expected that it may be reused to implement instruction scheduling for Patmos.

8.2 The DeepSpec Expedition in Computing

The DeepSpec Expedition in Computing [App+17] is a project that attempts to determine key methods and technologies to enable industrial-scale formal specifications of software and hardware. The main hypothesis of the DeepSpec project is that formal methods for both software and hardware are now so developed that widespread use should be encouraged, and, simultaneously, that formal methods will very soon be considered necessary in many domains where informal development methodologies

have previously been accepted. The project encompasses hardware architecture, compilers, operating systems, and some proof-of-concept applications. The DeepSpec also has a social objective to "spread the good news" about formal specifications among scientists, students, and industry. For this purpose, the project has published textbooks on formal methods [Pie+19a], programming languages [Pie+19b], algorithms [App18], and testing [LP18], as well as a number of experience reports (e.g. [Bre+18], [MAN17]).

The overall technical aim is to be able to compose independently developed proofs of correctness, so that large verified systems can (more) easily be built out of smaller reusable verified components. The DeepSpec project attempts to enable this by developing so-called *deep specifications* of each interface between components. The project defines deep specifications as specifications that are:

- Rich, meaning that they describe all behaviours in detail, even if they are very complex
- Two-sided, meaning that the implementation and the clients are both connected to the specification
- Formal, meaning that the specification is written in a (mathematical) notation with clear semantics to support automatic tools
- Live, meaning that the specification is connected via proofs to the implementation and the client code.

The specification of CompCert C is one such deep specification, since each transformation is proven correct in the semantics of the relevant intermediate languages, and thus CompCert can actually be used as a back-end for higher level tools.

Since one of the main ideas of the project is that components should be able to be developed separately, the project group is split into several subprojects across the participating universities. When all the subprojects are combined, the idea is that it is possible to compose the proofs from each subproject to obtain an "end-to-end" proof of correctness of the entire system. Some of the most relevant subprojects will be explained in more detail below.

8.2.1 The Princeton Verified Software Toolchain

The Verified Software Toolchain [App11] is the DeepSpec subproject that is most directly related to CompCert. It consists of a language and program logic called Verifiable C, a retargetable separation logic, and some other verified program analysis tools. The toolchain takes as its input a program written in Verifiable C (a subset of CompCert C), which can then be analysed using the various tools in the toolchain.

The most important part of the project is that the Verifiable C language is a subset of CompCert C, and can thus be compiled using CompCert. Additionally, the program analysis proofs from the toolchain can be composed directly with the

proof of the correctness of CompCert, thus creating a "stack" of proofs that is more powerful than CompCert by itself: not only can it be proven that the C program is correctly translated to assembly, but the correctness of the C program itself can also be proven, and the two proofs can be composed with no holes that must be trusted.

With a CompCert backend for Patmos, the Verified Software Toolchain should also work on Patmos with only minor obstacles.

8.2.2 Vellvm

The Vellvm project [Zha+12] (short for verified LLVM [LA04]) is a framework for proofs about transformations of programs written in LLVM's intermediate representation. Like CompCert, it provides a formal specification of the semantics of the intermediate representation language and its type system. Like CompCert, the specifications and program transformations are written in Coq.

Vellvm is essentially a competitor to CompCert, but the project is substantially smaller in scope, focusing on proving single transformations rather than the entire compilation process from end to end. Thus the project focuses not on complete correctness, but on ensuring correct code in the most complicated parts of the system.

8.2.3 Kami

Using the Verified Software Toolchain and CompCert, one can write C code, formally verify that it functions correctly, and compile it to assembly code which is proven to have the same semantics as the C code. Thus it is possible to certify that the generated assembly code functions correctly. However, there is still no guarantee that the actual processor executing the code is correct.

Kami [Cho+17] is a platform for specifying, implementing, and verifying hardware designs in a subset of the Bluespec language [Arv03]. Like the other DeepSpec projects, it is implemented in Coq. With Kami it is possible to implement and verify a formally specified instruction set architecture. If the implemented instruction set architecture is identical to (or at least compatible with) the architecture used for the chosen CompCert backend, it is possible to compose the proof of assembly correctness with the proof of processor correctness to obtain a proof that the hardware design will actually execute the code so that it functions correctly.

Kami can currently extract designs into "normal" Bluespec code, but the process of developing a formally verified Bluespec compiler that can turn the design into netlists is still in progress. Additionally, the project relies on conventional (unverified) place and route technology to turn the netlist into a physical design. Future work is needed to realize the goal of end-to-end proofs of correctness from high-level code to physical hardware.

8.2.4 CertiCoq

All the previously mentioned projects (as well as this project) are implemented in Coq. An obvious issue is then: can it be guaranteed that Coq itself is correct? While the logic of Coq is known to be consistent, there is still an issue of extracting the Gallina code into executables. For example, CompCert is extracted into OCaml, which is then compiled using an unverified OCaml compiler.

CertiCoq [Ana+17] is an ongoing project with the goal of eliminating this issue. The project is creating a verified, optimizing compiler for Coq, targeting machine language. CertiCoq is also implemented in Coq, and the basic principle of the compiler is to transform Gallina code into CompCert C light, which is then compiled further using CompCert. In this way, it is possible to create a verified extraction pipeline all the way from Gallina to machine language. A benefit of using CompCert is that all target architectures of CompCert can be targeted.

8.3 Formalisation of the Patmos pipeline

Patmos is designed to be time-predictable to make worst-case execution time analysis efficient. This includes the pipeline of Patmos, which is designed to preserve the timing of instructions. However, pipelines often introduce timing anomalies, i.e. a situation where a local worst-case timing does not lead to a global worst-case timing. It is possible to prove that pipelines are without timing anomalies using a formal model of the pipeline. This is done for a number of time-predictable pipelines, including Patmos, in [Jan+].

CHAPTER 9

Future work

The Patmos port of CompCert is not currently finished, so bringing it to a functional state would be a natural extension of this project.

Once a functional port of CompCert for Patmos has been developed, it is essential to thoroughly test the formal specification of the Patmos ISA to ensure that it is compatible with the actual behaviour of Patmos implementations. If this is not the case, either the compiler or the implementation must be changed for the compiler to produce semantically correct code. An even better option is to formally specify the intended functionality of the Patmos ISA and develop Patmos based on the formal specification in the future, so that the formal specification is the "source of truth", and implementations are tested instead of requirements.

To reap the full benefits of the time-predictability of Patmos, the compiler must generate code that fits the assumptions of the Patmos processor design, and assist the WCET analysis tool by generating information for, and making optimizations based on, the analysis.

For example, all functions in the generated code must be small enough to fit inside the method cache for good WCET analysability and performance. The compiler can accomplish this by splitting long functions in several parts to fit inside the cache.

The integrations with the WCET analysis tool include supplying high-level meta-information from the compiler to the analysis tool, feedback from the analysis tool to the compiler about optimization choices, and forwarding flow annotations from the source code through to compiler to be used in the analysis.

CompCert is an optimizing compiler, and Patmos has some optimization possibilities that are not feasible on "ordinary" processors. For example, the current port does not attempt to model the dual-issue nature of Patmos' pipeline, and thus the compiler can only generate half of the instructions that would otherwise be possible (though the actual performance gain of generating dual-issue code would be less than 100%, since not all program segments can be implemented by instructions in parallel).

Additionally, the compiler does not attempt to use the different memory and cache types of Patmos, nor does it attempt to place instructions in the delay slots of loads and multiplication instructions. Since the Patmos instruction set is fully predicated, another possible optimization is generation of single-path code, or at least code with less branches by utilizing predicates to "merge" code branches into one.

The CompCert compiler itself is not fully verified, but contains several modules that are not verified. Thus it would also be useful to extend the compiler in general

so that the entire compiler can be certified.

CHAPTER 10

Conclusion

The goal of this project was to implement a a backend for the CompCert compiler that allows it to translate C code into assembly code for the Patmos processor. This goal was not fulfilled, as the backend was only partially implemented. The partial implementation can not be used for actually compiling assembly in its current state, but it may be used as a foundation for implementing a functioning backend.

The partial implementation includes a reworked CompCert common language infrastructure and Mach language that avoids 64-bit floating point values, a formal definition of the Patmos assembly language syntax, and an almost fully implemented formalisation of the semantics of many instructions in the Patmos ISA. What still needs to be done before the compiler is fully functional is to finish formalising the semantics of Patmos, and to implement the translator that actually generates Patmos instructions from Mach instructions.

It is expected that the combination of a formally verified compiler and a time-predictable processor would, provided the implementation of the compiler was fully functional, enable users of the platform to implement safer embedded systems.

Since the backend was not implemented fully, it has not been possible to test the performance or correctness of the compiler port. It is expected that the performance of the compiler will be worse than existing tools if the backend were to be completed according to the plans laid out in this thesis, as the backend as designed does not implement any Patmos-specific optimisations.

Several directions of future work seem promising, chiefly among them, of course, to actually implement a fully functional backend. Once this is done, Patmos-specific optimizations and features may be implemented, and projects to fully certify both the compiler and the processor could be explored.

Bibliography

[18] *Road vehicles – Functional safety*. Standard. Geneva, CH: International Organization for Standardization, December 2018.

[99] *Programming languages – C*. Standard. Geneva, CH: International Organization for Standardization, December 1999.

[AK19] Jesse Alama and Johannes Korbmacher. "The Lambda Calculus". In: *The Stanford Encyclopedia of Philosophy*. Edited by Edward N. Zalta. Spring 2019. Metaphysics Research Lab, Stanford University, 2019.

[Ana+17] Abishek Anand et al. "CertiCoq: A verified compiler for Coq". In: *The Third International Workshop on Coq for Programming Languages*. CoqPL 2017. Paris, January 2017.

[App+14] Andrew W. Appel et al. *Program Logics for Certified Compilers*. New York, NY, USA: Cambridge University Press, 2014. ISBN: 9781107048010.

[App+17] Andrew W. Appel et al. "Position paper: the science of deep specification". In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 375.2104 (2017). DOI: 10.1098/rsta.2016.0331.

[App11] Andrew W. Appel. "Verified Software Toolchain". In: Lecture Notes in Computer Science ESOP 2011: 20th European Symposium on Programming.6602 (March 2011), pages 1–17.

[App18] Andrew W. Appel. *Verified Functional Algorithms*. Volume 3. Software Foundations. 2018. URL: https://softwarefoundations.cis.upenn.edu/current/vfa-current/index.html.

[App98] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998. ISBN: 0-521-58274-1.

[Arv03] Arvind. "Bluespec: A Language for Hardware Design, Simulation, Synthesis and Verification". In: *Proceedings of the First ACM and IEEE International Conference on Formal Methods and Models for Co-Design*. MEMOCODE '03. Washington, DC, USA: IEEE Computer Society, 2003, pages 249–. ISBN: 0-7695-1923-7. URL: http://dl.acm.org/citation.cfm?id=823453.823860.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. 1st edition. Addison-Wesley Publishing Company, 1986. ISBN: 0201100886.

[Bar91] Henk Barendregt. "Introduction to generalized type systems". In: *Journal of Functional Programming* 1.2 (1991), pages 125–154. DOI: 10.1017/S0956796800020025.

[BCT94] Preston Briggs, Keith D. Cooper, and Linda Torczon. "Improvements to Graph Coloring Register Allocation". In: *ACM Trans. Program. Lang. Syst.* 16.3 (May 1994), pages 428–455. ISSN: 0164-0925. DOI: 10.1145/177492.177575. URL: http://doi.acm.org/10.1145/177492.177575.

[Bre+18] Joachim Breitner et al. "Ready, Set, Verify! Applying Hs-to-coq to Real-world Haskell Code (Experience Report)". In: *Proc. ACM Program. Lang.* 2.ICFP (July 2018), 89:1–89:16. ISSN: 2475-1421. DOI: 10.1145/3236784. URL: http://doi.acm.org/10.1145/3236784.

[Bro15] Peter Brown. "How Math's Most Famous Proof Nearly Broke". In: *Nautilus* 24 (2015). URL: http://nautil.us/issue/24/error/how-maths-most-famous-proof-nearly-broke (visited on February 25, 2019).

[CH88] Thierry Coquand and Gérard Huet. "The calculus of constructions". In: *Information and Computation* 76.2 (1988), pages 95–120. ISSN: 0890-5401. DOI: https://doi.org/10.1016/0890-5401(88)90005-3. URL: http://www.sciencedirect.com/science/article/pii/0890540188900053.

[Chl13] Adam Chlipala. *Certified Programming with Dependent Types*. MIT Press, 2013. ISBN: 9780262026659.

[Cho+17] Joonwon Choi et al. "Kami: A Platform for High-Level Parametric Hardware Specification and Its Modular Verification". In: *Proceedings of the ACM on Programming Languages*. ICFP '17 24. ACM, September 2017. DOI: 10.1145/3110268.

[Chu32] Alonzo Church. "A Set of Postulates for the Foundation of Logic". In: *Annals of Mathematics* 33.2 (1932), pages 346–366. ISSN: 0003486X. URL: http://www.jstor.org/stable/1968337.

[Chu40] Alonzo Church. "A Formulation of the Simple Theory of Types". In: *The Journal of Symbolic Logic* 5.2 (1940), pages 56–68. ISSN: 00224812. URL: http://www.jstor.org/stable/2266170.

[Coq10] The Coq Development Team. *The Coq Reference Manual, version 8.9.0*. January 2010. URL: https://github.com/coq/coq/releases/download/V8.9.0/coq-8.9.0-reference-manual.pdf.

[Coq92] Thierry Coquand. "Pattern Matching with Dependent Types". In: *Proceedings of the 1992 Workshop on Types for Proofs and Programs*. Edited by Bengt Nordstrøm, Kent Petersson, and Gordon Plotkin. Båstad, June 1992, pages 66–79.

[Cur34] H. B. Curry. “Functionality in Combinatory Logic”. In: *Proceedings of the National Academy of Sciences of the United States of America* 20.11 (1934), pages 584–590. ISSN: 00278424. URL: http://www.jstor.org/stable/86796.

[Cur41] Haskell B. Curry. “The Paradox of Kleene and Rosser”. In: *Transactions of the American Mathematical Society* 50.3 (1941), pages 454–516. ISSN: 00029947. URL: http://www.jstor.org/stable/1990124.

[Deg+14] P. Degasperi et al. “A Method Cache for Patmos”. In: *2014 IEEE 17th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*. June 2014, pages 100–108. DOI: 10.1109/ISORC.2014.47.

[ER08] Eric Eide and John Regehr. “Volatiles Are Miscompiled, and What to Do About It”. In: *Proceedings of the 8th ACM International Conference on Embedded Software*. EMSOFT ’08. Atlanta, GA, USA: ACM, 2008, pages 255–264. ISBN: 978-1-60558-468-3. DOI: 10.1145/1450058.1450093. URL: http://doi.acm.org/10.1145/1450058.1450093.

[GA96] Lal George and Andrew W. Appel. “Iterated Register Coalescing”. In: *ACM Trans. Program. Lang. Syst.* 18.3 (May 1996), pages 300–324. ISSN: 0164-0925. DOI: 10.1145/229542.229546. URL: http://doi.acm.org/10.1145/229542.229546.

[Gim95] Eduardo Giménez. “Codifying Guarded Definitions with Recursive Schemes”. In: *Selected Papers from the International Workshop on Types for Proofs and Programs*. TYPES ’94. Berlin, Heidelberg: Springer-Verlag, 1995, pages 39–59. ISBN: 3-540-60579-7. URL: http://dl.acm.org/citation.cfm?id=646535.695850.

[Gir72] Jean-Yves Girard. “Interprétation Fonctionnelle et Élimination des Coupures de l’Arithmétique d’ordre Supérieur”. PhD thesis. Université Paris VII, June 1972.

[GTL03] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types*. Cambridge: Cambridge University Press, 2003. ISBN: 0-521-37181-3.

[How80] William A. Howard. “The formulae-as-types notion of construction”. In: *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*. Edited by Jonathan P. Seldin and J. Roger Hindley. Boston, MA: Academic Press, 1980, pages 479–490. ISBN: 978-0-12-349050-6.

[Iem16] Rosalie Iemhoff. “Intuitionism in the Philosophy of Mathematics”. In: *The Stanford Encyclopedia of Philosophy*. Edited by Edward N. Zalta. Winter 2016. Metaphysics Research Lab, Stanford University, 2016.

[Jan+] Mathieu Jan et al. “Formal Semantics of Predictable Pipelines: a Comparative Study”. Unpublished.

[Kah87] Gilles Kahn. “Natural Semantics”. In: *4th Annual Symposium on Theoretical Aspects of Computer Sciences on STACS 87*. Passau, Germany: Springer-Verlag, 1987, pages 22–39. ISBN: 0-387-17219-X. URL: http://dl.acm.org/citation.cfm?id=28220.28222.

[LA04] Chris Lattner and Vikram Adve. “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation”. In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*. CGO ’04. Palo Alto, California: IEEE Computer Society, 2004, page 75. ISBN: 0-7695-2102-9. URL: http://dl.acm.org/citation.cfm?id=977395.977673.

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

[Ler09a] Xavier Leroy. “A formally verified compiler back-end”. In: *Journal of Automated Reasoning* 43.4 (2009), pages 363–446. DOI: 10.1007/s10817-009-9155-4.

[Ler09b] Xavier Leroy. “Formal verification of a realistic compiler”. In: *Communications of the ACM* 52.7 (2009), pages 107–115. URL: http://xavierleroy.org/publi/compcert-CACM.pdf.

[LG09] Xavier Leroy and Hervé Grall. “Coinductive Big-step Operational Semantics”. In: *Inf. Comput.* 207.2 (February 2009), pages 284–304. ISSN: 0890-5401. DOI: 10.1016/j.ic.2007.12.004. URL: http://dx.doi.org/10.1016/j.ic.2007.12.004.

[LP18] Leonidas Lampropoulos and Benjamin C. Pierce. *QuickChick: Property-Based Testing in Coq*. Volume 4. Software Foundations. 2018. URL: https://softwarefoundations.cis.upenn.edu/qc-current/index.html.

[MAN17] William Mansky, Andrew W. Appel, and Aleksey Nogin. “A Verified Messaging System”. In: *Proc. ACM Program. Lang.* 1.OOPSLA (October 2017), 87:1–87:28. ISSN: 2475-1421. DOI: 10.1145/3133911. URL: http://doi.acm.org/10.1145/3133911.

[Mar10] Simon Marlow, editor. *Haskell 2010 Language Report*. 2010. URL: https://www.haskell.org/definition/haskell2010.pdf.

[Pau15] Christine Paulin-Mohring. “Introduction to the Calculus of Inductive Constructions”. In: *All about Proofs, Proofs for All*. Edited by Bruno Woltzenlogel Paleo and David Delahaye. Volume 55. Studies in Logic (Mathematical logic and foundations). College Publications, 2015. ISBN: 978-1-84890-166-7.

[Pie+19a] Benjamin C. Pierce et al. *Logical Foundations*. Volume 1. Software Foundations. 2019. URL: https://softwarefoundations.cis.upenn.edu/current/lf-current/index.html.

[Pie+19b] Benjamin C. Pierce et al. *Programming Language Foundations*. Volume 2. Software Foundations. 2019. URL: https://softwarefoundations.cis.upenn.edu/current/plf-current/index.html.

[Plo04] Gordon D. Plotkin. "A structural approach to operational semantics". In: *J. Log. Algebr. Program.* 60-61 (2004), pages 17–139.

[Pug91] William Pugh. "The Omega Test: A Fast and Practical Integer Programming Algorithm for Dependence Analysis". In: *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*. Supercomputing '91. Albuquerque, New Mexico, USA: ACM, 1991, pages 4–13. ISBN: 0-89791-459-7. DOI: 10.1145/125826.125848. URL: http://doi.acm.org/10.1145/125826.125848.

[Pus+13] P. Puschner et al. "The T-CREST approach of compiler and WCET-analysis integration". In: *16th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2013)*. June 2013, pages 1–8. DOI: 10.1109/ISORC.2013.6913220.

[Que88] Ruy J.G.B. de Queiroz. "A Proof-Theoretic Account of Programming and the Role of "Reduction" Rules". In: *Dialectica* 42.4 (1988), pages 265–282. ISSN: 00122017, 17468361. URL: http://www.jstor.org/stable/42970596.

[Rei+16] Alastair Reid et al. "End-to-End Verification of ARM Processors with ISA-Formal". In: *Proceedings of the 2016 International Conference on Computer Aided Verification (CAV'16)*. Edited by S. Chaudhuri and A. Farzan. Volume 9780. Lecture Notes in Computer Science 9780. Toronto, Canada: Springer Verlag, July 2016, pages 42–58. ISBN: 978-3-319-41539-0. DOI: 10.1007/978-3-319-41540-6_3.

[Rei16] Alastair Reid. "Trustworthy Specifications of ARM v8-A and v8-M System Level Architecture". In: *Proceedings of Formal Methods in Computer-Aided Design (FMCAD 2016)*. Mountain View, CA, USA, October 2016, pages 161–168. ISBN: 978-0-9835678-6-8. URL: https://alastairreid.github.io/papers/fmcad2016-trustworthy.pdf.

[Rey74] John C. Reynolds. "Towards a Theory of Type Structure". In: *Programming Symposium, Proceedings Colloque Sur La Programmation*. Berlin, Heidelberg: Springer-Verlag, 1974, pages 408–423. ISBN: 3-540-06859-7. URL: http://dl.acm.org/citation.cfm?id=647323.721503.

[SBM20] Cyril Six, Sylvain Boulmé, and David Monniaux. "Certified Compiler Backends for VLIW Processors". Unpublished. 2020.

[Sch+15] Martin Schoeberl et al. "T-CREST: Time-predictable multi-core architecture for embedded systems". In: *Journal of Systems Architecture* 61.9 (2015), pages 449–471. ISSN: 1383-7621. DOI: https://doi.org/10.1016/j.sysarc.2015.04.002. URL: http://www.sciencedirect.com/science/article/pii/S1383762115000193.

[Sch+18] Martin Schoeberl et al. “Patmos: A Time-predictable Microprocessor”. In: *Real-Time Systems* 54(2) (April 2018), pages 389–423. ISSN: 1573-1383. DOI: 10.1007/s11241-018-9300-4.

[Sch+19] Martin Schoeberl et al. *Patmos Reference Handbook*. March 2019. URL: http://patmos.compute.dtu.dk/patmos_handbook.pdf.

[Sch09] Martin Schoeberl. “Time-predictable Computer Architecture”. Candidate thesis. Technische Universität Wien, 2009. URL: https://www.jopdesign.com/doc/tparch.pdf (visited on February 25, 2019).

[Sco70] Dana Scott. *OUTLINE OF A MATHEMATICAL THEORY OF COMPUTATION*. Technical report PRG02. OUCL, November 1970, page 30.

[SK95] Kenneth Slonneger and Barry Kurtz. *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach*. 1st. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0201656973.

[Smo77] C. Smoryński. “The incompleteness theorems”. In: *Handbook of Mathematical Logic*. Edited by J. Barwise. Amsterdam: North-Holland Publishing Company, 1977, pages 821–866. URL: http://www.karlin.mff.cuni.cz/~krajicek/smorynski.pdf.

[Swa+16] Nikhil Swamy et al. “Dependent Types and Multi-Monadic Effects in F*”. In: *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, January 2016, pages 256–270. ISBN: 978-1-4503-3549-2. URL: https://www.fstar-lang.org/papers/mumon/.

[Tro91] A. S. Troelstra. “History of constructivism in the twentieth century”. In: *ITLI Prepublication Series for Mathematical Logic and Foundations*. ML-91-05. Amsterdam: Institute for Logic, Language and Computation, University of Amsterdam, 1991.

[Tur37] A. M. Turing. “Computability and λ-definability”. In: *Journal of Symbolic Logic* 2.4 (1937), pages 153–163. DOI: 10.2307/2268280.

[Yan+11] Xuejun Yang et al. “Finding and Understanding Bugs in C Compilers”. In: *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’11. San Jose, California, USA: ACM, 2011, pages 283–294. ISBN: 978-1-4503-0663-8. DOI: 10.1145/1993498.1993532. URL: http://doi.acm.org/10.1145/1993498.1993532.

[Zha+12] Jianzhou Zhao et al. “Formalizing the LLVM Intermediate Representation for Verified Program Transformations”. In: *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’12. Philadelphia, PA, USA: ACM, 2012, pages 427–440. ISBN: 978-1-4503-1083-3. DOI: 10.1145/2103656.2103709. URL: http://doi.acm.org/10.1145/2103656.2103709.

{2.7182818284}ϑφερτυθιοπσδφγηξκλ