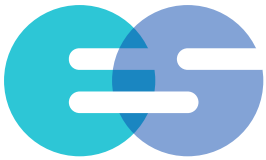


TUHH
Hamburg University of Technology

Hamburg University of Technology (TUHH)
Institute of Embedded Systems

Bachelor Thesis

Designing and Testing a Code Selector Ruleset for Jump-Statements and Pointer-Expressions targeting ARM Processors as part of a WCET-aware Compiler

Tobias Marschner

Hamburg, December 2019

First Examiner: Prof. Dr. Heiko Falk
Second Examiner: M. Sc. Dominic Oehlert

Statutory Declaration

I herewith declare that I have composed the present thesis myself and without use of any other than the cited sources and aids. Sentences or parts of sentences quoted literally are marked as such; other references with regard to the statement and scope are indicated by full details of the publications concerned. The thesis in the same or similar form has not been submitted to any examination body and has not been published. This thesis was not yet, even in part, used in another examination or as a course performance.

Place, Date

Signature (Tobias Marschner)

Abstract

To increase the versatility of the WCET-aware compiler (WCC), traditionally having targeted the Infineon TriCore architecture, implementing a native code selector for the ARM7 family of processors has been an ongoing process. This code selector performs tree pattern matching to translate its input and is generated automatically by the ICD-CG, a code selector generator. This thesis expands the ICD-CG's ruleset to enable translation of certain jump-statements (`goto`, `break` and `continue`), the `switch`-statement as well as expressions involving pointers found in ANSI-C, the WCC's source language.

Finally, a set of ANSI-C programs designed to specifically test for the above constructs is written and compiled with the WCC. The resulting assembly code is assessed with respect to correctness, code size and performance and is compared to assembly code generated by the `gcc`.

Table of Contents

1. Introduction and Motivation

The title of this thesis reads ”Designing and Testing a Code Selector Ruleset for Jump-Statements and Pointer-Expressions targeting ARM Processors as part of a WCET-aware Compiler”. This title is quite long and introduces several concepts, such as *code selection*, the *ARM processor architecture* and the notion of a *WCET-aware compiler*. Later chapters will examine these technologies closely and explain them in depth. However, a few fundamentals must be established first. This chapter aims to establish said fundamentals and provide the motivation for this thesis.

Essential to the targeted processor architecture and the WCET-aware C Compiler is the notion of *embedded systems* which will be explored first in section 1.1. Then, the focus will shift to the compiler infrastructure expanded in this thesis, the WCET-aware C Compiler. While chapter 2 will explain this compiler's structure and workflow in depth, section 1.2 will outline the motivation behind it. Section 1.3 will motivate this thesis itself, i.e. why the development of a custom code selector ruleset for this compiler is desirable, and give an overview of related works. Finally, section 1.4 will outline the contents of the following chapters.

1.1. Embedded Systems

Unless otherwise noted, the information presented in this section has been adapted from Peter Marwedel's *Embedded System Design* [1]. *Embedded Systems* are the key technology that stand at the heart of this thesis and they can be found everywhere in the real world.

For instance, modern cars rely on a variety of auxiliary electronic systems, such as air bag control systems, anti-braking systems (ABS), electronic stability programs (ESP) and more. Many consumer electronic devices, such as TVs with integrated receivers, handheld game consoles and smartphones fall under the umbrella of *embedded systems.* Medical devices such as a pacemaker or even an electric toothbrush present further examples. A traditional area of application for *embedded systems* is robotics, with special focus put towards the mechanical aspects.

This raises the question what exactly an *embedded system* is, and what common characteristics they often share. Peter Marwedel defines *embedded systems* as ”information processing systems

embedded into enclosing products“ [1]. The crucial common characteristic among these devices is their strong link to the real world, specifically with respect to time and concurrency. Edward A. Lee defines *embedded software* as ”software integrated with physical processes“ [2].

A related term often coming up when discussing *embedded systems* are *cyber-physical systems* (CPS). According to Lee, ”Cyber-Physical Systems (CPS) are integrations of computation and physical processes“ [3] and can therefore be understood to encompass *embedded systems* and the *physical environment* they are embedded in.

One of the most important characteristics of *cyber-physical systems* is **dependability**. This is due to the *safety-critical* nature of many CPS which is often the case in the automotive domain or the area of avionics. To be *dependable* refers to five key characteristics:

1. **Reliability** is the probability that a system will not fail.
2. **Maintainability** refers to the probability that a failing system can be repaired within a certain time-frame.
3. **Availability** is the probability that the system is available and is the result of a system's *reliability* and *maintainability*.
4. A system is considered **safe** if it will not cause any harm.
5. A system is considered **secure** if confidential data remains confidential and if authentic communication is guaranteed.

Of course, another crucial property *embedded systems* must satisfy is **efficiency**. Note that this can take on many different forms, from traditional goals such as *run-time efficiency*, *code size* or *cost* to properties that are much more important in *embedded systems*, such as *weight* or *energy efficiency*.

Most *embedded systems* are **dedicated towards a certain application**, since running multiple applications on a single system could compromise its *dependability*. Smartphones or multi-media solutions in cars present notable exceptions to this characteristic.

The common property *most important to this thesis*, however, is that many *cyber-physical systems* have to meet **real-time constraints**. Hermann Kopetz defines the instant in which a result must be produced as a *deadline*. A deadline is called *soft* if the result is useful even after the deadline has passed, otherwise it is called *firm*. A deadline is called a *hard deadline* if *severe consequences* could result from missing it, such as loss of life or grave environmental damage [4]. This point will be expanded upon in the next section.

1.2. Motivation behind the WCET-aware C Compiler

The compiler that will be extended as part of this thesis is the *WCET-aware C Compiler* of the *Institute of Embedded Systems* at the *Hamburg University of Technology*. From now on

this compiler will be referred to as the WCC. As the name implies, this compiler's defining characteristic is that it is aware of a program's *Worst-case execution time* (WCET), a term that will be explained in this section. Unless otherwise noted, all of the information presented in this section is adapted from Paul Lokuciejewski and Peter Marwedel's work on WCET-aware compilation techniques for real-time systems [5].

As was touched upon in the previous section, many *embedded systems* are *real-time systems*, meaning the system's correctness depends not only on the results of its computations, but also on the time at which they are produced [6]. One critical property when reasoning about a task's timeliness is its *worst-case execution time* (WCET) which denotes the task's longest execution time that can ever occur. It is generally impossible to precisely determine a program's WCET, since any task in practice exhibits a state space too large to explore through an exhaustive search.

One approach to determine the WCET of a program is through *simulation and measurements*. Estimates acquired through this technique are insufficient, however, since there is no guarantee that the task's actual WCET is not higher. Instead, *formal methods* are used to estimate a task's WCET.

It is generally impossible to estimate the WCET through formal methods, since the problem is reducible to the *halting problem*. However, a restricted set of programs fulfilling two special criteria can be analyzed statically: The program in question (1) must terminate and (2) recursion depths as well as loop iteration counts must be explicitly bounded.

Even then, a precise estimation of the WCET remains infeasible. Modern processors are equipped with a variety of features that will introduce a lot of variation to its timing behavior, such as superscalar pipelines, speculative execution or caches. Additionally, abstractions have to be made for a program's input and initial state.

Therefore, the WCET that can be estimated through formal methods presents an *upper bound*. The act of determining such a WCET-estimate is called *timing analysis*. For timing analysis to be reliable it must fulfill two criteria:

- **Safeness**: The estimated WCET *has to be higher or equal* to the actual WCET.
- **Tightness**: The difference between the estimated and the actual WCET should be as small as possible.

One crucial problem often encountered by programmers developing software for real-time systems is the lack of a timing model in the code. Moreover, due to a lack of software development tools that are timing-aware, the common practice when developing software for real-time systems is a *trial-and-error* approach. It usually consists of the following steps:

- Send the latest build through simulation to determine an estimation for the WCET from measurements. Since there is no guarantee that this estimate is *safe*, a *safety margin* is added on top of the measured WCET.
- The programmer has to change the source based on their *intuition*. There are no clear pointers for them what exactly should be changed in the source, and it is unclear what precise effect a change will have on the program's timing behavior.
- Repeat these two steps until the timing-behavior matches the specification.

This approach is slow due to its iterative nature, error-prone since there are no clear indicators for the programmer what to adapt and it is expensive, since the added safety-margin may increase the cost of hardware beyond what is actually required.

The *WCET-aware C Compiler* aims to solve these problems. Reliable timing analyses are tightly integrated into the compiler and are transparently and automatically invoked [7]. Thanks to its timing-awareness, it can reason about the timing-effects different modifications can have on the program. This is especially useful since many optimizations employed by compilers optimizing the *average-case execution time* (ACET) may, in fact, worsen a program's WCET. The WCC is able to determine whether an applied optimization would cause such an adverse effect.

This is the motivation behind the *WCET-aware C Compiler* that will be expanded as part of this thesis. A detailed explanation of the compiler framework will be provided in chapter 2.

1.3. Motivation for this Thesis and Related Work

One curious fact not mentioned before in this chapter is the fact that the WCC is already able to translate C sources to ARM assembly. To that end, code selection is *not* performed by the WCC's internal code selector. Instead, the WCC invokes the *GNU C Compiler*'s (`gcc`) ARMv4 code selector during compilation, passing the (potentially modified) C source along. The assembly code generated by the `gcc` is then parsed back into the WCC so that low-level timing analyses and optimizations can be performed on the generated code.

This is undesirable for a number of reasons. It is difficult to understand the `gcc`'s internal design philosophies as it performs code selection, especially with respect to memory and stack management. Additionally, the `gcc` does not provide the WCET-aware optimizations discussed in the previous section, and it does not integrate and connect well with the WCC's other components. That is why it has been an ongoing effort to develop a ruleset for the WCC's internal code selector that targets the ARM7 family of processors. This thesis is one of the many works that contribute to this effort.

In 2017 Abir Bouraffa expanded the WCC's already existing code selector framework to ensure compatibility with the ARM7 family of processors [8]. In the same year Janina Plog developed a ruleset translating arithmetic and logic operators, conditional statements, loops and function

calls [9]. In 2018 Paavo Becker designed rules to translate statements and expressions involving *composed types*, which specifically includes `struct`, `union` and array types [10]. Finally, Jan Runge developed the set of rules that ensure proper handling of type casts [11].

1.4. Structure of this Thesis

This chapter provided the introduction and motivation to this thesis, explaining key concepts such as *embedded systems* and the importance of the *worst-case execution time*.

The next two chapters, 2 and 3, will explain two key technologies in depth that are essential to the code selector ruleset: First, a detailed explanation of the WCC and its *intermediate representations* is given, before the latter chapter provides an overview of the ARMv4 architecture, the processor architecture targeted during this thesis.

Chapter 4 will explain the **ICD-CG**, the *code selector* framework in which the developed ruleset is embedded. Chapter 5 presents part one of the code selector ruleset, showcasing the rules developed for *jump* and `switch` *statements*. The large variety of rules developed for *pointer expressions* will be tackled in chapter 6, which also provides extensive explanations of the two essential auxiliary classes that were developed alongside the ruleset, `AddressModification` and `DerefInfo`.

Chapter 7 will test the developed ruleset with respect to *correctness* by utilizing the WCC's builtin *testbench*, as well as *code size* and *efficiency* with the aid of *simulations* and *static timing analyses*. Finally, chapter 8 concludes this thesis, looking back on what has been achieved and what remains to be done.

2. The WCET-aware C Compiler

The *WCET-aware C Compiler* is the compiler framework which builds the foundation for this thesis. It translates C sources to valid assembly code of various target architectures, ARMv4 being the target of interest in this thesis. Before diving into the WCC's specifics in section 2.3, including its *intermediate representations*, the following two sections establish foundations: Section 2.1 outlines the path from C sources to executable binaries while section 2.2 explains how a *compiler* operates in general.

2.1. Workflow for C Sources

C99, a standardized variant [12] of the C programming language, is the WCC's source language. When invoking a *C compiler* like the *gcc*, *clang* or even the WCC, more steps than just compilation itself take place. Aho et al. provide an overview of that workflow which is summarized here [13]. Figure 2.1 illustrates the process.

The **preprocessor** prepares every C source file for compilation, executing *preprocessor-directives* such as `#define` or `#include`. The **compiler** then translates each preprocessed C source individually, producing human-readable *assembly code* of the chosen target as output. This *assembly code* still contains several instructions that could not actually be executed in their current form such as label-based branches or subroutine calls. The **assembler** encodes all of the given instructions in the format of its target architecture and produces *machine code* packaged in so-called *object files*.

Up until this point every C source file has been processed *independently*. The **linker** glues all of the *object files* together, resolving any open references, and performs final adjustments on the code to ensure it can actually be run on the target architecture. Its output is an *executable binary*, which can for instance be encoded in the *Executable and Linkable Format (ELF)*.

2.2. Structure of a Compiler

Now that the general framework of how C sources are translated has been established, this section will take a closer look at the *compilation* step. This section, too, is based on the information provided by Aho et al [13].

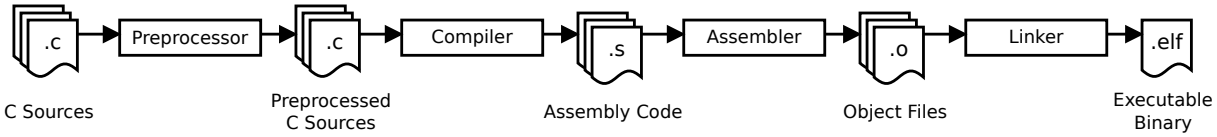


Figure 2.1.: Pipeline from C sources to executable binaries [13]

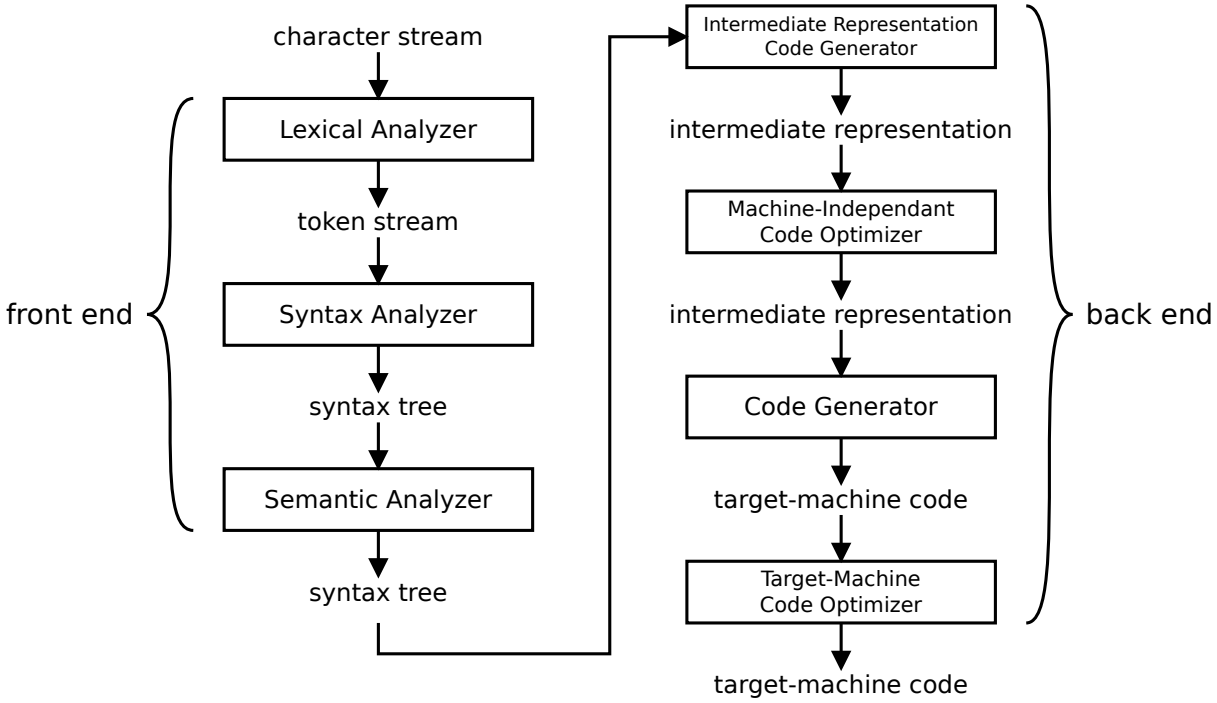


Figure 2.2.: Different phases of a compiler [13]

Figure 2.2 details the different phases of a compiler. Note that this pipeline is not definitive, rather presenting common components often found in practice. For instance, some compilers may opt to skip the *intermediate representation*, while others may even include multiple ones.

The first half of compilation consists of several analyses and is therefore often called the *analysis stage* or the *front end*. In the very first phase, a **lexer** transforms the input file, interpreted as a *stream of characters*, into a *stream of tokens*. Tokens can for instance be identifiers, reserved keywords, built-in types, operators, separators, terminators, etc...

This *token stream* is still a linear representation of the source file. The **parser** transforms the *token stream* into a *syntax* ***tree***, ordering the tokens after the source language's rules. If the parser cannot parse the given *token stream* according to those rules, a *syntactic error* has been detected (e.g. a missing semicolon).

Even if the **parser** runs successfully, there may still be errors in the source that make compilation impossible, so-called *semantic errors*. A *semantic analyzer* looks out for such errors and may optionally annotate the syntax tree with additional information inferred during this phase. In C mismatched types, assignments to constants or function calls with too many arguments present just a slice of possible *semantic errors* that can be found.

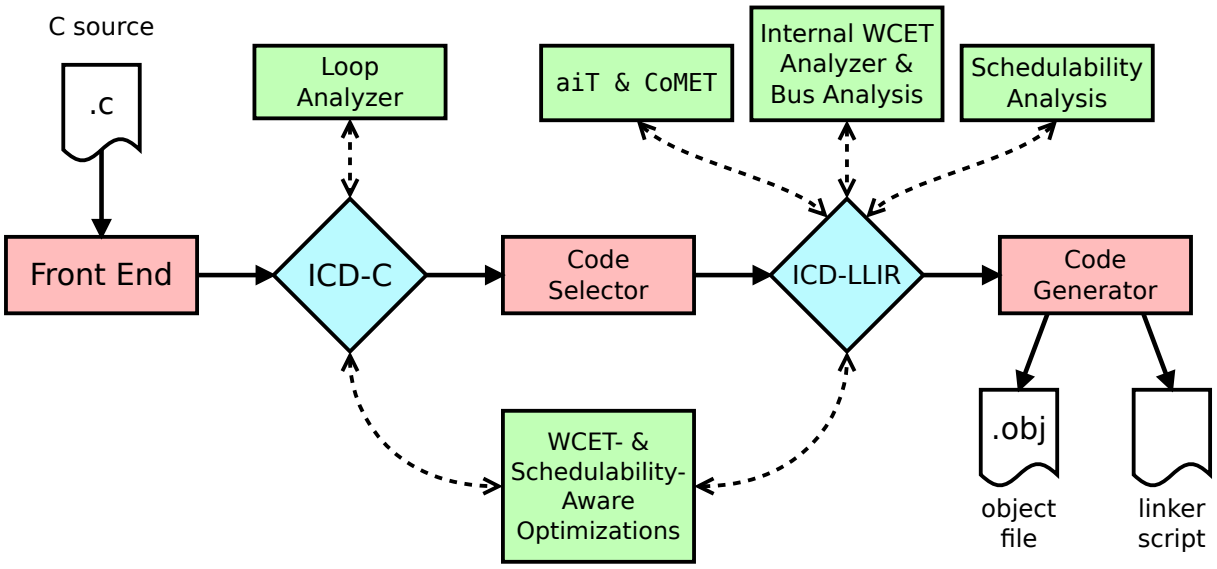


Figure 2.3.: Workflow of the WCC [7]

A so-called **symbol table** is constructed during the *analysis stage*, with every phase contributing information it discovered about the source code. For example, the *symbol table* collects the type, scope and name of identifiers such as variables or constants. This information can then be used by phases in the *back end*, where knowledge of a variable's type or a function's argument list is invaluable. Even phases in the *front end* itself will build on and use the information discovered by previous phases.

After the *analysis stage* a **code generator** translates the *(annotated) syntax tree* to a new representation. In figure 2.2 the input is first translated into an *intermediate representation*. What exactly this representation looks like and why it was chosen depends entirely on the goals and purpose of the compiler as well as its source and target language. As stated earlier, a compiler may skip *intermediate representations* entirely or even implement multiple ones. Often, *intermediate representations* are chosen in such a way that certain **optimizations** can be applied that are difficult to perform on the *syntax tree* or the final *target-machine code*.

Eventually a **target-machine code generator** will produce code in the compiler's target language. **Target-machine optimizations** present the final touch before the compiler concludes its job.

2.3. Structure of the WCC

The general workflow of the *WCET-aware C Compiler* (WCC) is shown in figure 2.3 and roughly follows the model presented in the previous section. Information in this section is largely based on the overview given by Oehlert et al. [7], while Lokuciejewski and Marwedel provide a deeper description of the *intermediate representations* [5].

Shown in red are those transformations usually found in a typical compiler, including the **front end**, the WCC's **code selector** as well as the **code generator**. The blue diamonds represent the WCC's *intermediate representations*, consisting of the *high-level intermediate representation* **ICD-C** that is very close to the C source and the *low-level intermediate representation* **ICD-LLIR** that is very close to target-machine assembly code. These two *intermediate representations* (IRs) will be discussed in depth in the next two subsections. Finally, the processes in green boxes connected with dotted lines indicate those features of the WCC that make it *WCET-aware*, including static timing analyses and optimizations.

As outlined in the previous section, the **front end** consists of a *lexer*, *parser* and *semantic analysis step* and, in the WCC's case, outputs the **ICD-C** IR. The terms **code selector** and **generator** given in the figure sound equivalent, but serve very different purposes in the WCC. The **code selector** uses *tree pattern matching* to translate the **ICD-C** IR to the **ICD-LLIR**. The **code generator**'s task is simpler: It merely extracts the assembly code packaged inside of the **ICD-LLIR** and outputs it to a text file. Since the **code selector** is essential to the ruleset developed in chapters 5 and 6, inserting instructions and utilizing properties of the ARMv4 target architecture, it will be discussed later, in chapter 4.

The **ICD-C** allows the annotation of code with so-called *flow facts*. As defined by Kirner in his PhD thesis, *flow facts* "give hints about the possible [control flow paths] of a program" [14]. For instance, they give information about a recursive function's *maximal depth* or indicate an upper bound for how many times a loop's body statement is going to be executed. These *flow facts* can be defined *implicitly* by the program's source code. To this end, the **loop analyzer** can infer aforementioned *loop bounds* from many loop constructs automatically. *Flow facts* can also be defined explicitly by the programmer, using `#pragma`-directives.

A brief motivation for these *flow facts* has been provided in section 1.2. They are necessary to allow *static timing analyzers* to terminate, since the problem of estimating the WCET would otherwise be *undecidable*. One such *static timing analyzer* integrated into the WCC is `AbsInt`'s `aiT` [15]. The WCC also comes equipped with `CoMET`, a cycle-accurate instruction set *simulator* developed by *Synopsys* [16]. Both are tightly integrated with the WCC, which generates the respective configuration files and invokes the tools automatically and transparently [7]. The analyses' results are annotated back to the **ICD-LLIR**. A feature called the *back-annotation* even allows for information associated with the **ICD-LLIR** to be transferred to the **ICD-C** IR. For instance, precise timing information derived from the **ICD-LLIR** can then be used by the **ICD-C** to determine whether a high-level optimization such as *loop unrolling* would have positive or adverse effects.

Next to `aiT` the WCC also features its own internal WCET analyzer for ARM7TDMI-based multicore systems [17]. Various bus arbitration policies, private and shared memories as well as hierarchical cache structures are supported in this analysis framework. *Schedulability analyses* are also incorporated into the WCC, supporting both fixed and dynamic scheduling. These

cooperate with the WCC's timing analyses, taking overhead of the scheduler and context switches into account.

2.3.1. High-Level Intermediate Representation: ICD-C

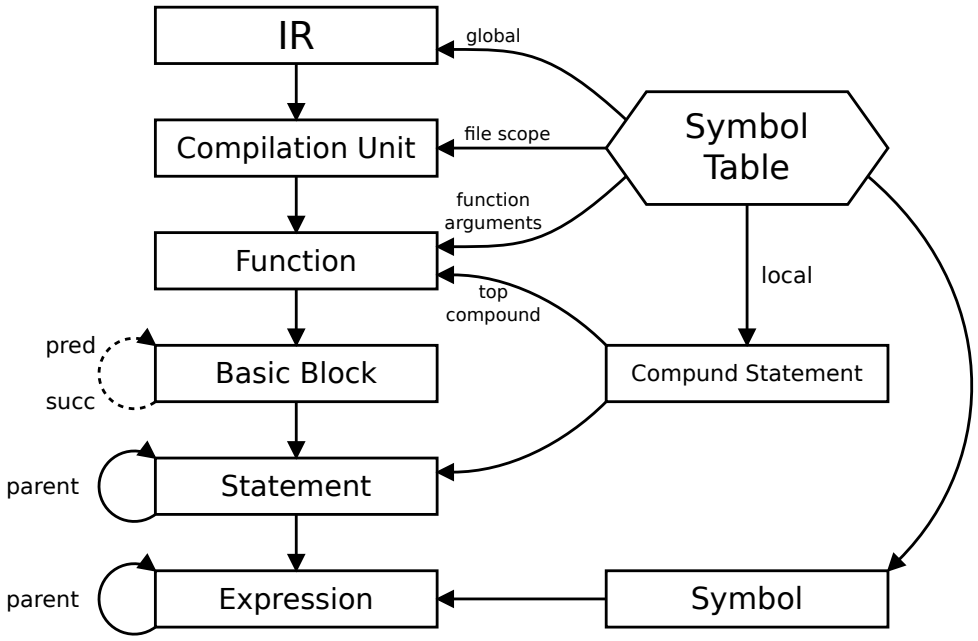


Figure 2.4.: Simplified class-model of the **ICD-C** IR [5]

The **ICD-C** *intermediate representation* is very close to the analyzed syntax tree of the C source code, to the extent that it can be transformed back to plain C source code. The official website gives an overview [18], while Lokuciejewski et al. provide a more detailed description [5].

Figure 2.4 gives an overview of the **ICD-C**'s structure. The **IR** as a whole may consist of multiple **compilation units** representing individual source files to be processed. Multiple **functions** can be given inside of a single source file, each equipped with a **top compound statement**. In C, multiple statements can be *compounded* into a single statement by encapsulating them in *curly braces* (`{}`). When defining a function, these curly braces are mandatory and the statements given within this **top compound statement** constitute the *function body*.

Each function is then split into a list of **basic blocks**, each containing a list of **statements**. Since the concept of **basic blocks** can be better illustrated with lower-level components, its explanation is postponed to the next subsection. **Statements** in C are usually terminated with *semicolons* (`;`) and can contain multiple **expressions**. *Control-flow statements* like `if` or `for` contain other **statements** that they control. Therefore, a notion of hierarchy can be established between **statements**.

The **symbol table** discussed in the previous section provides information about the various identifiers present in each level of figure 2.4's hierarchy. Some variables may be valid in the

```
1 int abs( int x )
2 {
3   if ( x < 0 )
4     return -x;
5   else
6     return x;
7 }
```

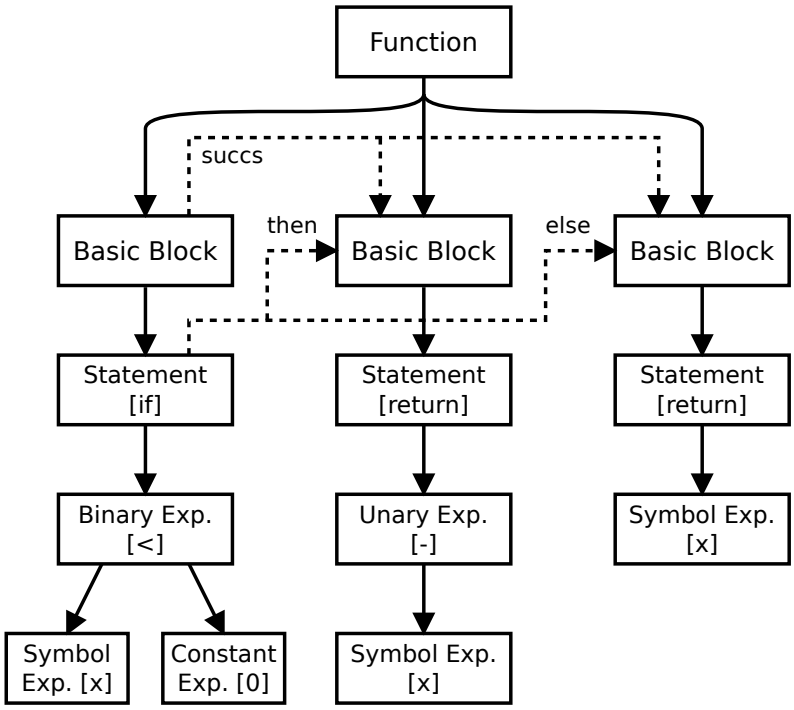


Figure 2.5.: Simplified **ICD-C** representation (right) for a small C function (left)

global or *file-wide* scope, while others can only be used in specific local scopes.

The **ICD-C** framework provides several optimizations and analyses. *Control flow* analyses such as determining *dominance relations* or *reachability analysis* as well as *data flow* analyses like identifying *def/use chains* are supported. The WCC comes equipped with many common ACET optimizations such as *dead code elimination*, *common subexpression elimination* or *function inlining*.

Finally, **ICD-C** supports two features that aid the WCC in being WCET-aware. First, it is able to interpret pragma preprocessor-directives (`#pragma ...`) which allow the programmer to manually annotate the source. For example, loop bounds that cannot be inferred from the source can be specified by the programmer, allowing the WCC's static timing analyses to incorporate these so-called *flow facts* in their calculations, giving more accurate results. Additionally, *persistent objects* can be attached to IR constructs to store any kind of information associated with them. In the WCC they are used to attach timing data to basic blocks.

Example

Figure 2.5 illustrates what the **ICD-C** IR would look like for the small example function given. Note that the presented model omits *a lot* of information present in the actual IR for the sake of simplicity.

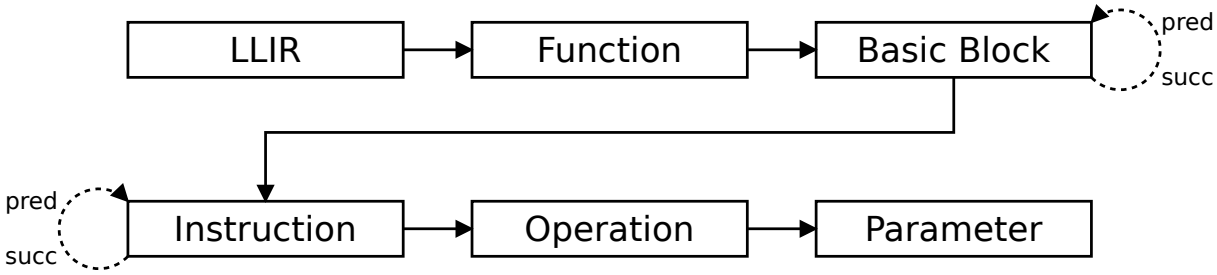


Figure 2.6.: Simplified class-model of the **ICD-LLIR** IR [5]

The top-level block considered here is the *function* `abs` that consists of three *basic blocks*. In this case, each of the *basic blocks* only has one *statement* attached. Both *return-statements* as well as the *if-statement* contain an expression. These expressions usually contain multiple expressions themselves, forming an *expression tree*. This can be seen for the *conditional expression* `x < 0` where the *binary operator* `<` is the parent to its two child expressions, `x` and `0`.

Dotted lines represent the interdependencies present in the IR. Every *basic block* points to its set of successors, whereas the *if-statement* references the *basic blocks* control flow would reach depending on the result of the *condition expression*. These interdependencies leverage the **ICD-C** IR above a simple syntax tree and help tremendously when designing the *code selector*'s ruleset in chapters 5 and 6.

2.3.2. Low-Level Intermediate Representation: ICD-LLIR

The **ICD-LLIR** is the flexible, generic *low level intermediate representation* used within the WCC. Again, the official website provides an overview [19], while Lokuciejewski et al. provide a more detailed description summarized here [5].

By itself, the **ICD-LLIR** is target-agnostic, meaning multiple target-architectures can be represented and only when supplied with a *target description* the **ICD-LLIR** becomes processor-specific. Code generated by the *code selector* that translates the **ICD-C** IR to the **ICD-LLIR** is always processor-specific. In other words, the **ICD-LLIR** is very close to the final assembly code, but this code is packaged in generic components that are the same across architectures. Figure 2.6 gives an overview of these components.

The **LLIR**-object itself represents a single compilation-unit / assembly file and can contain multiple **functions**, also called *subroutines*. Similar to the *high-level* **ICD-C** a **function** consists of one or more **basic blocks**.

A **basic block** represents a list of instructions which are *always* executed sequentially. *Control-flow* will only ever branch at the edges of a basic block, never within one. Consequently, any *conditional* or *unconditional jump* will always be the final instruction in its basic block, whereas any label will always reside at the very beginning of its basic block.

Each **instruction** can contain multiple **operations**. This allows the **ICD-LLIR** to support *very long instruction word (VLIW)* architectures, where *instruction-level parallelism* is implemented at compile-time, encoding multiple operations as part of a single instruction. Since ARMv4 is not a VLIW architecture, in this thesis every **instruction** contains exactly one **operation**.

Every **operation** is defined by its *mnemonic*, also called the *instruction code*, specifying which of the processor's instructions to perform. The operation's behavior is further specified by its **parameters**. Parameters include *immediate constants*, *labels*, processor-specific *operators* and *registers*. Note that *registers* are a special case and are managed by the `LLIR_Register` class, only referenced in a parameter through a *pointer*.

Many of the **ICD-C** IR's analyses and optimizations are available for the **ICD-LLIR**, too. *Lifetime*, *def/use chain* as well as *reachability* and *dominance analyses* are available. *Constant folding*, *dead code elimination* and *peephole optimizations* present a selection of supported optimizations. One of the most important optimizations performed on the **ICD-LLIR** is *register allocation*, which is discussed below.

Register Allocation

The *register allocator* is a component tightly linked with the **ICD-LLIR**. Code emitted by the *code selector* uses a mix of *physical* and *virtual registers*. *Physical registers* are those actually available on the target architecture, whereas *virtual registers* are *placeholders* that are mapped to *physical* ones at a later point.

The crucial advantage is the *infinite* number of *virtual registers* available. The rules of the *code selector* do not have to worry about how to distribute processed values between the limited register space and the stack memory. Instead, it can use *virtual registers* everywhere and leave aforementioned task to the *register allocator*.

This *register allocator* can, for example, utilize a *graph-coloring* heuristic involving liveness information of the virtual registers to determine which *virtual register* to assign to which *physical register* and where to insert *load and store* instructions that temporarily *spill* values on to the stack in case the physical register space alone does not suffice.

3. The ARMv4 Architecture

The target of this work is the ARMv4 architecture, implemented in the ARM7TDMI family of microprocessors. First, a general introduction of the architecture with its defining features and characteristics will be presented. Afterwards, the so-called *Addressing Modes* will be explained in depth, as they are of special significance for the pointer expression ruleset presented in chapter 6.

All of the architectural information presented within this chapter is taken from the *ARM7TDMI Technical Reference Manual* [20] as well as the *ARM Architecture Reference Manual* [21].

3.1. Overview

ARMv4 is a 32-bit microprocessor architecture based on *Reduced Instruction Set Computer* (RISC) principles. RISC machines are characterized by the simplicity in their design and architecture, introducing a *semantic gap* between high-level languages like C and the processor's *instruction set architecture* (ISA). Their counterpart are *Complex Instruction Set Computers* (CISC) that often require a complex decoding logic to interpret their instructions [22].

3.1.1. Memory Properties

A common property of RISC machines also present in ARMv4 is the *load-store architecture*. This means that memory can only be accessed through *load*, *store* or *swap* instructions. For example, if one wants to perform an addition on a value in memory, the value has to first be loaded, then modified and finally stored back – it cannot be modified directly. Additionally, the ARM7TDMI processors follow the *Von-Neumann* model where instructions and data reside in the same address space [23].

Natively, ARMv4 supports three data-types:

- 32-bit values called *words* that should always be aligned to four-byte boundaries when stored in memory.
- 16-bit values called *halfwords* that should always be aligned to two-byte boundaries when stores in memory.
- 8-bit values called *bytes*.

The ARMv4 architecture has a total of 31 general-purpose registers and 6 status registers. The range of available registers changes depending on the processor's current *mode*. A mode switch occurs for example when an exception or interrupt is triggered. In the standard non-privileged operating mode, called user operating mode, 16 general purpose registers (`r0` through `r15`) as well as the *Current Program Status Register (CPSR)* are accessible.

The CPSR contains information about condition code flags, the current operating mode and which interrupts are currently enabled or disabled. There are four *condition code flags* in the CPSR: N, Z, C and V standing for "Negative", "Zero", "Carry" and "oVerflow" respectively. Certain instructions, e.g. *subtraction* [`SUB`] or *comparison* [`CMP`], can set these bits based on the result of their operation.

Of the 16 general-purpose registers, five registers fulfill a special role:

- `r15` / `pc` – The *program counter* always points at the next instruction to be executed and is continually incremented.
- `r14` / `lr` – The *link register* contains the return address of the current subroutine and is automatically overwritten when calling a *branch and link* [`BL`] instruction.
- `r13` / `sp` – The *stack pointer* points to the current end of the stack and is vital for storing *local* variables.
- `r12` / `ip` – The *intra procedure call scratch register* is reserved for use by the linker [24].
- `r11` / `fp` – The *frame pointer* points to the beginning of the current stack-frame.

The remaining 11 registers are used as variable / scratch registers inside a subroutine. Note that only `r15` and `r14` are treated specially by the *Instruction Set Architecture (ISA)*. The special roles of `r13`, `r12` and `r11` are merely the result of convention.

3.1.2. Instruction Set

The ARMv4 architecture offers two instruction sets: The default is the *ARM instruction set* where every instruction has a fixed length of 32 bits. The *Thumb instruction set* is a subset of the ARM instruction set with a fixed instruction length of 16 bits, sacrificing versatility in exchange for smaller code size. In this thesis, Thumb instructions will not be taken into closer consideration – all rules designed for the code-generator will emit 32-bit ARM instructions. Thanks to the CPSR many instructions support *conditional execution*, only performing their operation if a previous instruction met certain conditions.

The ARM instruction set can be roughly divided into six categories. Three of these are not important for this thesis and are omitted here. The relevant classes are introduced below, with example-snippets shown at the end of this section.

Branch instructions manipulate the *control-flow* by manually changing the value in the *program counter* [`pc`]. A simple *branch* [`B`] jumps to another location in memory by applying a

```
1 bl foo
```

Call subroutine `foo`

```
1 mov r0, #50
2 sub r1, r0, #8
```

Load 50 in `r0`, then subtract 8 and store result (42) in `r1`.

```
1 ldr r0, [sp, #16]
2 add r0, r0, #4
3 str r0, [sp, #16]
```

Add 4 to a value located in memory at `sp`+16.

Figure 3.1.: Example ARMv4 assembly snippets for *branch instructions* (left), *data-processing instructions* (middle) and *load and store instructions* (right)

relative offset of at most ±32MB. The *branch and link* [BL] instruction additionally writes the value of the current *program counter* [pc] minus 4 bytes to the *link register* [lr], which corresponds to the address of the instruction succeeding the *branch and link* [BL][1]. This is often used to call *subroutines* as the address inside the *link register* [lr] can be used to return from the call.

Data-processing instructions perform calculations with general-purpose registers. That can mean arithmetics such as *addition* [ADD] or *subtraction* [SUB], logic operations like *bitwise AND* [AND] or *bitwise OR* [ORR], comparison operations such as *compare* [CMP] or *test equivalence* [TEQ] or even the simple register-to-register copy operation *move* [MOV].

Load and store instructions form the final category discussed here, housing the *load*, *store* and *swap* instructions. These build the bridge between the processor's register set and the addressable memory since instruction from other categories cannot access the addressable memory by themselves.

Loads generally load a value from memory into a register, whereas *stores* perform the opposite. In both cases the type must be specified (*word*, *halfword* or *byte*) and when loading the signedness of the type is important as well. If the source type is smaller than 32-bits it is either *sign-* or *zero-extended* when loaded into the 32-bit register, depending on its signedness.

Finally, there are also *load multiple* [LDM] and *store multiple* [STM] instructions available that can be used to load or store multiple (or all) general-purpose registers with a single instruction.

Figure 3.1 presents three snippets of example ARMv4 assembly code, showcasing the three different classes of instructions just discussed. The leftmost snippet shows a simple *branch and link* [BL] instruction, calling a subroutine called `foo`. The snippet in the middle of the figure illustrates the usage of two *data-processing* instructions. In line 1 the immediate constant 50 is loaded into register `r0`. Then, line 2 subtracts the value 8 from the contents of register `r0` and writes the result into register `r1`. Finally, the rightmost snippet illustrates the usage of *load and store instructions*. First, the value located at `sp + 16` is loaded into register `r0`. The second

[1] Due to the 3-stage pipeline of the ARMv4 architecture, the *program counter* [pc] is "ahead" of the currently executed instruction by 8 bytes. Therefore, the address of the instruction succeeding the currently executed one is located at the *program counter* [pc] *minus* 4 bytes.

Syntax	Brief description
`#<immediate>`	An immediate operand directly encoded as part of the instruction.
`<Rm>`	A register operand, just like `<Rn>`.
`<Rm>, <shift> #<shift_imm>`	A register operand that is shifted by a constant number of bits.
`<Rm>, <shift> <Rs>`	A register operand that is shifted by the number of bits specified in `<Rs>`.

Table 3.1.: Different forms of the `<shifter_operand>` (`RRX` omitted)

line then increments the freshly loaded value by 4. The last line stores the modified value back to the location pointed to by `sp + 16`.

3.2. Addressing Modes

While subsection 3.1.2 gave an introduction to the instruction classes relevant to this thesis, one crucial question has not been answered so far: How exactly do these instructions specify their operands and what limitations apply?

The answers lie within the five *Addressing Modes* defined in the ARMv4 architecture, each detailing how certain classes of instructions can specify their operands. The following two subsections will take a closer look at *Addressing Modes 1, 2* and *3* as they are key in designing the pointer expression ruleset presented in chapter 6.

3.2.1. Addressing Mode 1 - Data-processing Operands

Addressing Mode 1 is used for all *data-processing instructions* which have been introduced earlier. These instructions generally comply with the following syntax:

```
<opcode>{<cond>}{S} <Rd>, <Rn>, <shifter_operand>
```

First, `<opcode>` describes the operation to be performed (i.e. `ADD`, `SUB`, ...), `<cond>` is an optional flag indicating conditional execution and the `S`-flag indicates whether this specific instruction should update the CPSR's conditional flags. `<Rd>` denotes the destination register, i.e. where to store the result of the operation. `<Rn>` is the first operand and is always a register. The second operand, however, can be given in multiple ways, most of which are shown in table 3.1.

Note that the immediate operand's value range is limited. It must be an 8-bit value optionally rotated by an even number of bits. The `<shift>` parameter denotes one of four different shift modes: Logical shift left (`LSL`), logical shift right (`LSR`), arithmetic shift right (`ASR`) or rotate right (`ROR`).

```
1 mov r0, sp
2 mov r1, #5
3 mov r2, #4
4 mul r1, r2, r1
5 add r0, r0, r1
```

Without *immediate shift*

```
1 mov r0, sp
2 mov r1, #5
3
4
5 add r0, r0, r1, lsl #2
```

With *immediate shift*

Figure 3.2.: Two ARMv4 assembly snippets achieving the same goal. The right one uses *shifted register operands*, the left one does not.

Example

The example given in figure 3.2 demonstrates the versatility granted by *Addressing Mode 1* thanks to its *shifted register operands*. Both assembly snippets realize the same calculation, where one uses *shifted register operands* while the other does not. First, both snippets initialize registers `r0` and `r1`. The actual goal is to multiply the contents of `r1` by 4 and add the result on top of `r0`.

The left snippet accomplishes this goal without using *shifted register operands*. The factor `#4` is first loaded into `r2`, then multiplied with `r1` and this intermediate result present in `r1` is finally added on top of `r0`. Note that *multiplication* [`MUL`] is an outlier within the *data processing instructions* because it does **not** support any *Addressing Mode*. Both operands have to be registers, necessitating the immediate load in line 3.

The right snippet is able to achieve the aforementioned goal with a single instruction. The second register operand `r1` is shifted left by two bits which is equivalent to a multiplication with factor 4. This intermediate value is then added on top of `r0`.

3.2.2. Addressing Modes 2 and 3 - Load and Store Operands

Load and store instructions specify their operands according to *Addressing Modes 2* and *3*. Depending on the data type loaded or stored, either *Addressing Mode 2* or *3* applies. For both cases, the syntax adheres to the following template:

```
{LDR|STR}{<cond>}{<variant>} <Rd>, <target_address>
```

During a *load* `<Rd>` is the destination register whereas during a *store* it specifies the source. Again, `<cond>` allows to specify conditional execution. Depending on the chosen `<variant>` either *Addressing Mode 2* or *3* applies, allowing different specifications of the `<target_address>`.

Before diving into the options for the `<target_address>`, table 3.2 gives an overview of the different *load* and *store* variants, detailing which *Addressing Mode* they belong to.

Variant	Explanation	Addr. Mode
`LDR` `STR`	Load / Store a *word* (32-bit value).	Mode 2
`LDRT` `STRT`	Load / Store a *word* (32-bit value). Acts like a user mode access even when in privileged mode.	post-indexed Mode 2

`LDRH`
`LDRSH`
`STRH`
	Load and zero-extend a *halfword* (16-bit value).
	Load and sign-extend a *halfword* (16-bit value).
	Store a *halfword* (16-bit value).
		
		Mode 3

`LDRB`
`LDRSB`
`STRB`
	Load and zero-extend a *byte* (8-bit value).
	Load and sign-extend a *byte* (8-bit value).
	Store a *byte* (8-bit value).
		Mode 2
		Mode **3**
		Mode 2

`LDRBT` `STRBT`	Load and zero-extend / Store a *byte* (8-bit value). Acts like a user mode access even when in privileged mode.	post-indexed Mode 2

Table 3.2.: Different variants of *load and store instructions*

Indexing	Offset	Syntax	Addr. Mode
—	Immediate	`[<Rn>, #+/-<offset>]`	2 & 3
	Register	`[<Rn>, +/-<Rm>]`	2 & 3
	Scaled Register	`[<Rn>, +/-<Rm>, <shift> #<shift_imm>]`	2
Pre-indexed	Immediate	`[<Rn>, #+/-<offset>]!`	2 & 3
	Register	`[<Rn>, +/-<Rm>]!`	2 & 3
	Scaled Register	`[<Rn>, +/-<Rm>, <shift> #<shift_imm>]!`	2
Post-indexed	Immediate	`[<Rn>], #+/-<offset>`	2 & 3
	Register	`[<Rn>], +/-<Rm>`	2 & 3
	Scaled Register	`[<Rn>], +/-<Rm>, <shift> #<shift_imm>`	2

Table 3.3.: Syntaxes of `<target_address>`

The `T`-variants (`LDRT`, `STRT`, `LDRBT`, `STRBT`) are of no further interest in this thesis because all code emitted by the compiler will operate in *non-privileged operating modes*.

Table 3.3 showcases the different syntaxes usable for the `<target_address>`. The final column indicates whether the specific option is available in *Addressing Modes 2* and *3* or whether it is exclusive to *Addressing Mode 2*.

Mandatory for every syntax is the register `<Rn>` which contains the *base address*. It is therefore impossible to directly access a purely immediate memory location. The nine available options can then be grouped by their *offset type* and *indexing*.

Three different *offset types* are available. The *immediate offset* adds or subtracts an immediate value to or from the *base address*. This value can be `#0` too, directly accessing the memory pointed to by `<Rn>`. In *Addressing Mode 2* `<offset>` is a 12-bit value, allowing a modification

```
1  // r4 contains the address to an int-array with 100 elements.
2  // An int is 4 bytes large, making the array 400 bytes long.
3  ...
4  mov r0, #0                    // Initialize loop counter.
5  looplabel:
6  ldr r1, [r4, r0, lsl #2]      // Load from the int-array.
7  sub r1, r1, #1                // Decrement the member.
8  str r1, [r4, r0, lsl #2]      // Store the result back to the array.
9  add r0, r0, #1                // Increment the loop counter.
10 cmp r0, #100                  // Test loop condition and ...
11 blt looplabel                 // ... repeat loop if r0 < 100.
12 ...
```

Listing (3.1) ARMv4 assembly snippet decrementing every member of an `int` array

of up to ±4095 bytes, whereas it is an 8-bit value in *Addressing Mode 3*, allowing a modification of up to ±255 bytes.

Alternatively, the *base address* can be offset by another register `<Rm>`, which is either added to or subtracted from the *base address* register `<Rn>`.

Finally, similar to the *shifted register operands* of *Addressing Mode 1*, the offset register `<Rm>` can optionally be shifted by an immediate value before it is added to or subtracted from the *base address* register `<Rn>`. The options for `<shift>` are the same as described in *Addressing Mode 1*. Note that this *shifted register offset* is exclusive to *Addressing Mode 2*.

There are also three different forms of *indexing*, all available in both *Addressing Mode 2* and *3*. All of the offsets described above perform an addition or subtraction (optionally combined with a shift) on the *base address*. The *indexing mode* determines whether this modification happens *before* or *after* the memory access, and whether the newly calculated address should be *written back* to the *base address* register `<Rn>`. This way, *address modification* and *memory access* can be combined into a single instruction, which can for instance be used when translating expressions like `*p++`. This feature will be made use of extensively in the ruleset designed for pointer expressions which will be presented in chapter 6.

The default option is to calculate the offset *temporarily*, use that temporary address for the memory access and then discard it. In *pre-* and *post-indexing* the newly computed address is written back to the *base address* register `<Rn>` instead of being discarded. In *pre-indexing* the newly computed address is used for the memory access, whereas in *post-indexing* only the *base address* without any offset is used.

Example

Listing 3.1 presents an example where the flexibility of *Addressing Mode 2* and its *shifted register offset* shines bright. The snippet presents a loop in which all members of an `int`-array are

decremented by 1.

`r0` is the loop index counting upwards from 0 to 99 and `r4` is the array's *base address*, pointing to the first member of the array. Thanks to the immediate shift applied in lines 6 and 8 the loop counter `r0` can be used directly to access the array's `int` members. The loop counter does not have to be shifted / multiplied by an additional instruction, nor is a second loop counter necessary that would track the array address.

4. ICD-CG: An OLIVE-compatible Tree Pattern Matcher

This chapter will explain the **ICD-CG**, the WCC's *code selector* that translates the *high-level intermediate representation* **ICD-C** to the *low-level intermediate representation* **ICD-LLIR**. [25] Section 4.1 will outline the **ICD-CG**'s basic mode of operation, before section 4.2 will highlight some of the syntactic details that will become relevant when developing the ruleset later in chapters 5 and 6. Finally, section 4.3 will present an example rule and explain it.

The **ICD-CG** is not actually a code selector by itself, but rather a so-called *code generator generator*. The **ICD-CG** takes a *set of rules* as its input and produces the actual *code generator* as output. This *code generator* is then integrated into the WCC's pipeline, taking the **ICD-C** IR as input and producing the target's assembly code packaged in the **ICD-LLIR** as output. As explained in subsection 2.3.2 this **ICD-LLIR** is still *virtual*, meaning it uses *virtual* instead of *physical registers* in most places.

To actually perform translation the **ICD-CG** uses an approach called *tree pattern matching*. This technique was first described and practically implemented by Aho, Ganapathi and Tjiang in an implementation called **twig** [26]. Two years later Fraser et. al. adopted many of **twig**'s ideas in a less flexible but faster *tree pattern matcher* called **burg** [27], which eventually received an update in the form of **iburg** [28]. In 1992 Tjiang expanded **twig** with advancements made by **iburg**, improving pattern matching speeds while still allowing for generalized costs. This new framework is called **OLIVE** [29].

This history of these *tree pattern matchers* is important because the **ICD-CG**'s syntax is **OLIVE**-compatible. Thereby, a lot of **OLIVE**'s ideas and concepts translate over to the WCC's *code selector*. This section aims to provide an explanation of **OLIVE**'s / the **ICD-CG**'s functionality and syntax. Note that since Tjiang's original technical report on **OLIVE** [29] was not accessible to the author at the time of writing, the presented information has instead been adapted from section A.1 of Guido Araujo's PhD thesis [30].

4.1. Fundamentals

Code selection in **OLIVE** is defined by the specified *ruleset*. Each of these rules matches against a **tree**, a certain pattern of **terminals** and **nonterminals**, and replaces it with a single **nonterminal**. The **ICD-C** IR's *syntax tree* constitutes **OLIVE**'s starting point, where every node of the *syntax tree* is a **terminal**.

Ideally, this tree of **terminals** can – through repeated application of **OLIVE**'s rules – be transformed into a single **nonterminal**. At this point **OLIVE** could successfully *tree pattern match* the input. If certain constructs cannot be matched by any of **OLIVE**'s rules, *code generation* fails.

Every rule has a so-called **action**-part associated with it. This part inserts the actual **LLIR** instructions and manually calls the **action**-parts of the **nonterminals** it matched. More accurately, it invokes the **action**-parts of the rules that produced said **nonterminals**.

Invoking the **action**-part of that single, final **nonterminal** will subsequently invoke the **action**-parts of the **nonterminals** it matched, and so forth. In other words, invoking the **action**-part of that single, final **nonterminal** will actually generate the code for the matched input *syntax tree*.

Often, **OLIVE** is able to find multiple *matchings* for a given input *syntax tree*. To this end, every rule has a **cost**-part associated with it as well. The cost a rule returns not only reflects the cost of this specific rule, but also incorporates the cost of every previous rule that had been applied to the matched **terminals** and **nonterminals**. In other words, the costs stack up during matching until the single, final **nonterminal** contains the cost of *the entire matching*.

With this information, **OLIVE** will attempt to find a *matching* of the given input *syntax tree* that is *minimal* with respect to this cost. If multiple matchings for a (sub)tree present itself, **OLIVE** will choose the one with the lower associated cost. What **cost** actually means is entirely within the hands of the implementation. Common metrics include the estimated execution time of inserted instructions, code size or simply the number of inserted instructions.

This is **OLIVE**'s basic model of operation. Note that the *input tree* the **ICD-CG** processes is not the *syntax tree* of the entire source file, but rather a *subtree* of the *syntax tree* corresponding to an individual **ICD-C** *statement*. Therefore, the *tree pattern matcher* is invoked multiple times during compilation to translate the source's statements.

4.2. Details of a Rule

The **ICD-CG**'s rules adhere to the following syntax:

nonterminal : **tree** { **cost** } = { **action** };

Nonterminal refers the nonterminal that represents this rule and **tree** to the pattern of **terminals** and **nonterminals** to be matched. **Cost** and **action** are blocks of C++ code (the language the WCC itself is implemented in) that compute the cost or perform code generation respectively.

The pattern **tree** can be one of the following, each option separated by a pipe (|):

nonterminal | **terminal** | **terminal** (**tree_list**)

Tree_list is a comma-separated list of further **tree** symbols. The first two options process an individual symbol, either allowing for the further transformation of a **nonterminal** or the processing of a single **terminal**. However, a lot of nodes in the input *syntax tree* have one or more *child-nodes* attached to them. Sometimes it is desirable to match an entire *subtree* with a single rule. The third option allows for such a specification, by not only mentioning a single **terminal** but also constraining the makeup of its *child-nodes* with further **tree**-specifications.

Inside of the **cost** and **action**-parts special, **OLIVE**-specific identifiers can be used that start with a `$` sign. This allows the included C++ code to interact with the pattern matcher directly, for instance to retrieve **costs** of child-nodes or invoke **action**-parts manually. The three most important identifiers for this theses are explained below.

- `$n` refers to the `n`th node in the **tree** pattern. `$0` always refers to the rule itself, while `$1` refers to the first **terminal** or **nonterminal** specified in the rule's **tree** symbol. If a *subtree* was specified, its nodes are numbered using *pre-order* traversal.
- `$cost[n]` retrieves the cost of the `n`th node. The identifier can also be used to set the cost of the rule itself with `$cost[0]`.
- `$action[n](a, b, ...)` manually calls the **action**-part of the `n`th rule. `a, b, ...` are the *parameters* passed to the **action**-part which are discussed in more detail below.

Both **terminals** and **nonterminals** have to be defined before they can be used. **Terminals** can be defined with a `%term` statement. However, the **ICD-C** framework already provides a list of **terminal** definitions corresponding to the different components of the **ICD-C** IR. Therefore, no **terminals** have to be defined as part of this thesis.

Nonterminals can be defined using a `%declare` statement which adheres to the following syntax:

```
%declare<return type> nonterminal<arguments>;
```

```
1  %declare< LLIR_Register* > reg<void>;
2
3  reg: tpm_BinaryExpPLUS( reg, reg )
4  {
5    // Cost includes the two child-nodes and a single ADD instruction.
6    $cost[0] = $cost[2] + $cost[3] + CT( INS_ADD_32 );
7  }
8  =
9  {
10   // Create a new virtual register for the result.
11   LLIR_Register* result = INSTRUCTIONS->CreateRegister( "" );
12
13   // Evaluate the two operands. (the child-nodes)
14   LLIR_Register *lhsReg = $action[2]();
15   LLIR_Register *rhsReg = $action[3]();
16
17   // Generate the ADD instruction.
18   INSTRUCTIONS->insertADD( result, lhsReg, rhsReg, $1->getExp() );
19
20   // Return the LLIR_Register* in which the addition's result is stored.
21   return result;
22 };
```

Listing 4.1: Example rule in **OLIVE** syntax, taken and edited from the WCC's source, originally developed by Janina Plog [9]

This declaration statement sheds light on a few special properties of **nonterminals** not previously discussed. First, **nonterminals** are represented by a C++ type, labeled `return type` above. A value of this type is returned by the **action**-part of every rule representing this **nonterminal**. For instance, a **nonterminal** representing integral constants that are up to 32 bits wide could use `int32_t` as its return type.

Moreover, every **nonterminal** specifies a list of `arguments`, which can also be left empty. These *arguments* have to be provided every time the **action**-part of a rule representing this **nonterminal** is invoked.

4.3. Example

Listing 4.1 presents one of the many rules found in the WCC's code selector ruleset, edited for brevity. This rule was originally developed by Janina Plog as part of her project thesis. [9] The code will now be examined step-by-step to understand what this rule's purpose is and what an **OLIVE**-rule looks like in practice.

Line 1 declares the used **nonterminal** `reg`, which is represented by the `LLIR_Register*` type in C++. As the name implies, this nonterminal represents *registers*, both *physical* and *virtual*. Since **action**-parts for this **nonterminal** do not require any arguments, `void` is specified in the second pair of angled brackets.

In line 3 the rule's definition starts with the **nonterminal** it represents, `reg`, as well as the **tree** pattern it matches. Specifically, it matches the **terminal** `tpm_BinaryExpPlus` that has two child-nodes which are both `reg` **nonterminals**. As the name implies, this **terminal** represents *binary addition*, like in `a + b`. Since both operands to be matched have to be represented by the `reg` **nonterminal**, this rule has to perform addition on two *registers*. Note that this implementation assumes *integer addition* is to be performed between the two registers, *floating point addition* is intentionally omitted here.

In the first pair of curly brackets `{}` the cost of the rule is calculated. In line 6 the cost of the rule itself (`$cost[0]`) is defined as the cost of evaluating both register operands (`$cost[2] + $cost[3]`) as well as the cost of inserting the *addition* [`ADD`] instruction. In the WCC's *code selector* framework the macro `CT` is provided which returns the cost incurred when inserting the specified instruction. In this case, that instruction is the 32-bit wide `ADD` instruction (`INS_ADD_32`).

The second pair of curly brackets after the equality sign (`=`) specifies the rule's **action**-part. In it, four steps are performed:

In line 11 a new *virtual register* is created. `INSTRUCTIONS->CreateRegister( "" )` calls an internal API that creates a new *virtual register* in the current **LLIR** function and returns a pointer to it. This pointer is stored in the `result` variable.

In lines 14 and 15 the **action** parts of the register operands are called. Since the nodes of the matched expression tree are numbered using *pre-order* traversal, they have the numbers 2 and 3 respectively. The `reg` **nonterminal** does not define any *arguments*, thereby none have to be passed when invoking the **action**-parts. Pointers to the evaluated register operands are then stored in the variables `lhsReg` and `rhsReg`, referring to *left-hand side register* and *right-hand side register* respectively.

In line 18 the internal API is called again to insert an *addition* [`ADD`] instruction. The `LLIR_Register*` for both the destination as well as the operands are passed to this helper function, along with the `IR_Exp*` corresponding to the **terminal** (`$1->getExp()`). The latter provides additional context to the helper function and is not of further interest here.

Finally, the pointer to the virtual register containing the result of this addition is returned, allowing any construct that contains this expression to use the result.

5. Designing the Ruleset for Jump and switch Statements

This and the upcoming chapter present the deep dive into the core of this thesis: The designed code selector ruleset. The chapter will begin with the rules for *jump statements* `break`, `continue` and `goto` in section 5.1 before moving to the `switch` statement in section 5.2. Finally, chapter 6 is solely dedicated to the ruleset matching *pointer expressions*, with thorough explanations of the *auxiliary classes* that were developed alongside it.

5.1. The Ruleset: Jump Statements

Section 6.8.6 of the C99 standard [12] lists four different jump statements, all of which are *unconditional* jumps, meaning if control-flow reaches this statement, the jump is always performed:

- `goto` *identifier* `;`
- `continue ;`
- `break ;`
- `return` $expression_{opt}$ `;`

Note that `return` statements are already implemented in the WCC's internal ARMv4 *code selector* and are therefore not of further interest in this thesis. They were originally implemented by Janina Plog as part of her project work [9].

The `goto` statement always jumps to the *label identifier* specified within the statement. This *label* has to be located in the same enclosing function.

The `continue` statement can only appear inside of or as the *loop body*. As a reminder, *loops* in C are `for`, `while` and `do-while` statements. The `continue` statement causes a jump to the end of the *loop body*. That means in `while` and `do-while` loops the *controlling expression* is evaluated next, whereas in a `for` loop the *reinitialization expression* is evaluated next.

The `break` statement can only appear in or as a `switch` body or *loop* body. It always terminates the smallest enclosing `switch` or *loop* statement, jumping outside of it.

5.1.1. ICD-C Interface

All three jump statements are represented by one and the same **terminal**, `tpm_JumpStmt`. This **terminal** has no child-nodes, even in the case of a `goto` statement. The second **terminal** of interest here is `tpm_LabelStmt`, which represents one of the labels a `goto` statement can branch to.

In this case, the **ICD-C** framework takes care of the bulk of the work when it comes to translating *jump statements*. Every statement in the **ICD-C** has a member function called `getImplicitJumpBasicBlock()`. If the statement this function is called on is the *last statement* of its basic block and only has *one* succeeding basic block, the method returns said successor.

Since *jump statements* influence the control flow, they always terminate the basic block they are contained in. Moreover, since every *jump statement* represents an *unconditional jump*, they must have *one and only one* succeeding basic block, the destination of the jump. Therefore, `getImplicitJumpBasicBlock()` can be used to determine the basic block that control flow has to branch to. This method can be used regardless of the type of jump statement, i.e. whether it is a `continue`, `break` or `goto` statement.

Sorting statements into their respective basic blocks is a task performed by the **ICD-C**. This means the *code selector* does not have to perform any further action when processing a *label statement*. It marks the destination of `goto` jumps, but no code is inserted at a label statement by itself.

5.1.2. OLIVE Rules

Figure 5.1 presents the source code of the **OLIVE** rules used to match the two previously discussed **terminals**, with debug code and comments removed. The `stmt` **nonterminal** represents complete **ICD-C** statements and is usually the single, final **nonterminal** returned by the pattern matching process. Since both **terminals** represent entire statements, terminated by semicolons, they transform their **tree** to the `stmt` **nonterminal**. **Action**-parts of this **nonterminal** do not take any arguments, nor do they return anything.

As mentioned before, no code has to be inserted for the *label statement*. Therefore, its **cost** is 0 and the **action** part does not insert any instructions.

The *jump statements* will always insert an *unconditional branch*. As explained in subsection 3.1.2, the *branch* [`B`] instruction is used to jump to another label in the assembly code. Since it is inserted for every *jump statement*, and is the *only* instruction inserted, the cost of a single *branch* [`B`] represents the **cost** of this rule. `CT( INS_B_32 )` looks up the cost of a 32-bit wide *branch* [`B`] instruction in a cost table already part of the WCC's ARMv4 code selector framework.

```
1  stmt: tpm_JumpStmt
2  {
3    $cost[0] = CT( INS_B_32 );
4  }
5  =
6  {
7    stringstream ss;
8    ss << $1->getStmt()->getImplicitJumpBasicBlock();
9    string block_label = CODESEL->getBlockLabel( ss.str() );
10   INSTRUCTIONS->insertB( OPER_AL, block_label );
11 };
12
13 stmt: tpm_LabelStmt
14 {
15   $cost[0] = 0;
16 }
17 =
18 {
19   // Empty.
20 };
```

Figure 5.1.: **OLIVE** rules for *jump* and *label statements*, slightly edited

The rule's **action**-part is a bit cryptic. In line 7 a temporary *stringstream* of C++'s *standard library* (`std::stringstream`) called `ss` is created. When referring to a **tree** node with the `$n` syntax, an `IR_TreeElem*` is returned. Method `getStmt()` returns the `IR_Stmt*` of the node, or a `nullptr` if it is not a statement.

Since *jump statements* are always statements, `$1->getStmt()` will yield a pointer to the `IR_Stmt` object representing the node. Then `getImplicitJumpBasicBlock()` is used to acquire the `IR_BasicBlock*` pointing to the jump's destination. Finally, this pointer is fed into the stringstream, where the pointer's address is stored as a string of hexadecimal letters.

The reason for this odd conversion is the method `getBlockLabel()` of the *code selector* API which is invoked in line 9. (`CODESEL->getBlockLabel( ... )`) This method expects a string representation of an `IR_BasicBlock`'s pointer and returns the label of the corresponding **LLIR** basic block. If no label exists yet, the method takes care of generating one and associating it with the **ICD-C** basic block.

Finally, line 10 inserts the actual *branch* [`B`] instruction, passing two parameters: First, `OPER_AL` is an operator indicating the condition under which the instruction is to be executed, in this case *always*. The second parameter is the label to branch to, which was determined in the preceding line.

5.2. The Ruleset: `switch` Statement

Section 6.8.4 of the C99 standard [12] defines the properties of *selection statements*, with section 6.8.4.2 giving details for the `switch` statement. As the name implies, *selection statements* select among a set of statements depending on the value of a *controlling expression*. The `switch` statement adheres to the following syntax:

`switch (` *expression* `)` *statement*

The *controlling expression* has to be of *integer type*. Moreover, *integer promotion* is performed on the *controlling expression*, meaning any types with a rank lower than that of (`unsigned`) `int` are implicitly converted to `int` or `unsigned int`. Specifically, this affects `char`, `signed char`, `short` and `unsigned short`.

In the WCC's ARMv4 code selection framework, *integer types* wider than 32 bits are currently not supported. Therefore, the only two types that currently need to be considered for the *controlling expression* are `int` and `unsigned int`. Since ARMv4 is a native 32-bit architecture, both types are 32 bits wide.

The *switch body* statement can contain multiple `case` labels that specify an *integer constant* which is of the same type as the *controlling expression*. The value of the *controlling expression* is compared against these constants, and if a match (equality) is found, control flow will branch to that `case` label. Therefore, no two `case` labels are allowed to specify the same constant.

Additionally, the programmer can specify a `default` label which is used as a fallback destination. If no `case` label matches the *controlling expression*'s value, control flow will branch to the `default` label. If no `default` label is given, the *switch body* will be skipped completely.

5.2.1. ICD-C Interface

The **ICD-C** framework defines three **terminals** that are relevant for `switch` statements: `tpm_SwitchStmt`, `tpm_CaseStmt` and `tpm_DefaultStmt`.

The latter two represent the `case` and `default` labels of the `switch` statement. Since both are effectively label statements, the reasoning presented in the previous section still applies: No instructions have to be inserted for the labels themselves, and the **cost** associated with them is 0.

The `tpm_SwitchStmt` **terminal** is of critical importance, however. It represents a `switch` statement, with the *controlling expression* being its only *child node*.

5.2.2. OLIVE Rules

Two rules have been designed for the `tpm_SwitchStmt` **terminal**, the difference lying in their child-nodes. The first implementation matches against a *value register* (`reg`) as its child node, whereas the other matches against a *constant integer expression* (`tpm_IntConstExp`) directly.

Rule matching against the `reg` nonterminal

First the rule matching against *variable controlling expressions* is presented. This refers to the scenario in which the `switch` statement's controlling expression, the child-node of the `tpm_SwitchStmt` **terminal**, is a `LLIR_Register*` identified by the `reg` **nonterminal**. Figure 5.2 presents a flowchart illustrating the workflow of this rule's action part which is discussed next.

The **action**-part begins by invoking the **action**-part of the child-node, thereby acquiring its `LLIR_Register*` in the process. Next, a loop is executed for every `case`-label present in the `switch` statement's body. The rule checks whether the constant integer specified by the `case`-label can be encoded as an immediate operand in an instruction using *addressing mode 1* (compare with subsection 3.2.1).

If the constant can be encoded as an immediate operand, a single *test equality* [`TEQ`] instruction is inserted. This instruction tests two operands for equality and updates the CPSR's status flags accordingly. Otherwise the constant has to first be loaded with a *move* [`MOV`] instruction and optionally up to three additional *logical OR* [`ORR`] instructions, in case the operand requires many set bits. Then, a *test equality* [`TEQ`] instruction is inserted comparing the contents of the just loaded case constant with those of the child-node.

Afterwards, a *conditional branch* [`BXX`] is inserted. There are a variety of conditional codes that can be used, in this case we wish to execute the branch only if the previous test for equality was positive. This conditional check can be encoded with the `EQ` suffix.

This process is repeated for every `case`-label, ultimately resulting in a chain of equality checks with subsequent conditional branches. When all of the `case`-labels have been processed, the final *unconditional branch* [`B`] is inserted. If a `default`-label is present this branch will target said label, otherwise it will branch to the *basic block* that succeeds the `switch`-statement as a whole.

The **cost**-part of this rule mirrors the structure presented in figure 5.2, but instead of inserting instructions it merely accumulates the cost of the instructions that would be inserted. Of course, the **cost** also includes the cost of the child-node. Unlike many of the rules for the pointer expressions presented later in subsection 6.5, no worst-case estimation is used here and the cost calculated by this rule's **cost**-part is accurate.

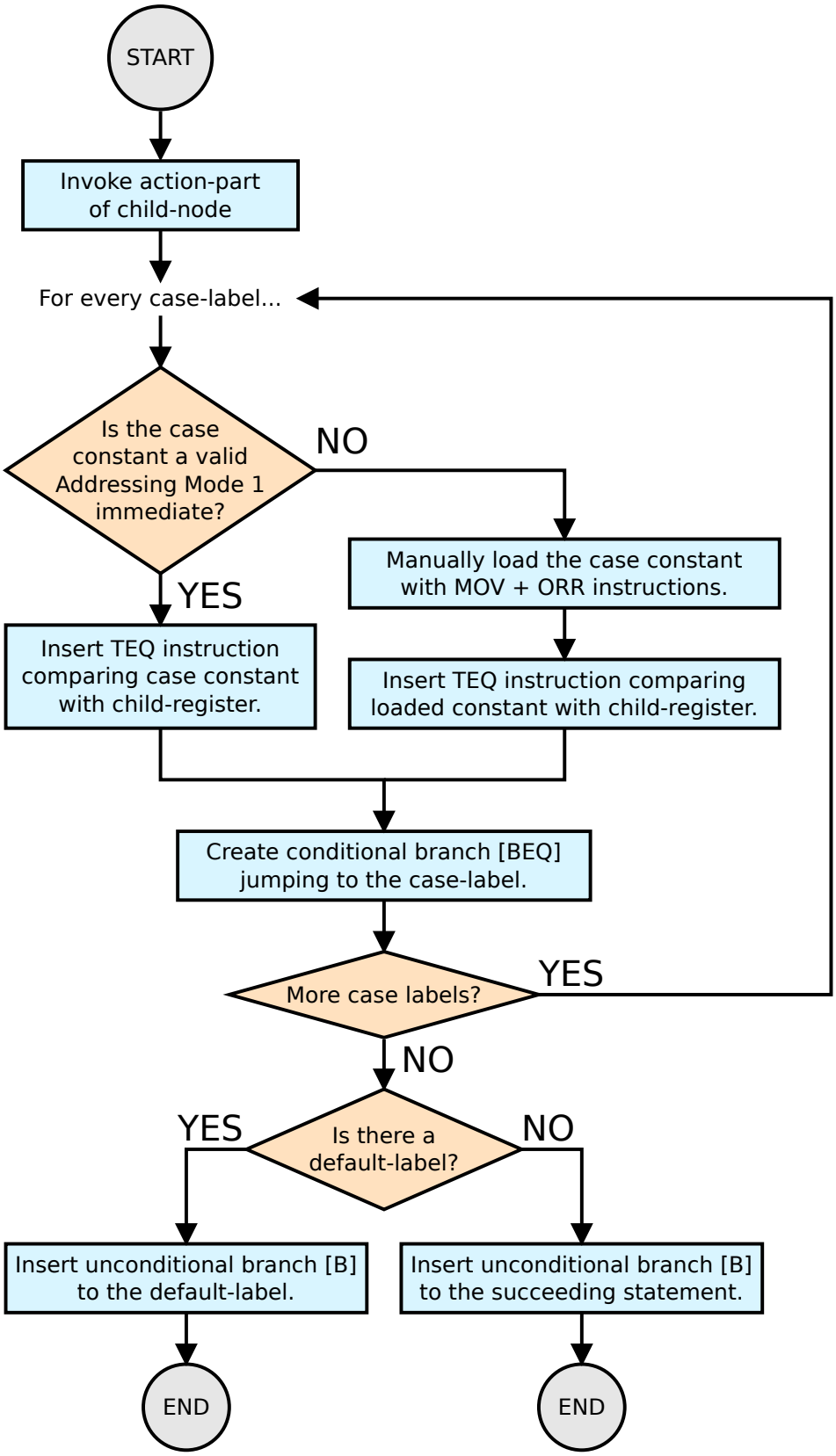


Figure 5.2.: Flowchart describing the operation of the `switch`-rule

Rule matching against the `tpm_IntConstExp` terminal

There is another rule matching against the `tpm_SwitchStmt` **terminal**. This one matches against a *constant controlling expression*, with the terminal representing *constant integer expressions* (`tpm_IntConstExp`) matched as the child-node. While it is not particularly sensible to write a `switch`-statement specifying code for various different cases, only to then supply a constant in the controlling expression, it is nonetheless a scenario that can be considered.

Since the controlling expression is constant, the question of which `case`-label to target can be evaluated at compile-time rather than runtime, essentially boiling the `switch`-statement down to a simple `goto`. That is exactly what the **action**-part of this rule does. It evaluates whether there is a matching `case`-label for the given constant integer. If there is, it creates an *unconditional jump* [B] to that `case-label`. Otherwise, it creates an *unconditional jump* [B] to the `default`-label, and if that does not exist, an *unconditional jump* [B] to the succeeding *basic block*. The **cost**-part of this rule always consists of a single *unconditional branch* [B] as well as the cost of its child-node.

6. Designing the Ruleset for Pointer Expressions

Chapter 4 presented an introduction to the **ICD-CG** code selection framework, while chapter 5 began explaining the ruleset designed for this thesis. This chapter dives into the most difficult and complex set of C constructs that had to be tackled: *Pointer expressions*.

First, the term *pointer expressions* will be properly defined in section 6.1, laying out which actual expressions are included under that umbrella term and detailing their corresponding **ICD-C terminals**. Section 6.2 will then provide a proper overview of all the **nonterminals** relevant to the developed rules, something which has not been done in sections 5.1 and 5.2 since the set of used **nonterminals** there was very small.

Sections 6.3 and 6.4 will explain the two auxiliary classes developed alongside the ruleset, highlighting key properties and public member functions. The actual rules will then be discussed in section 6.5 and section 6.6 will present a *tree pattern matching* example for a simple C statement.

6.1. Handled Expressions and their ICD-C Terminals

The term *pointer expressions* as given in the title of this thesis refers to a large set of expressions. Table 6.1 lists the various expressions that will be tackled as part of this thesis, next to their corresponding **terminals** as defined by the **ICD-C** framework and an example. Note that *function pointers* are **not** included in the scope of this thesis.

First up are basic expressions. It may seem trivial to point out, but *pointer symbols* have to be loaded just like any other symbol. The two essential operators when thinking about pointers are the *indirection operator* that uses an asterisk (`*`) and the *address operator* that uses an ampersand (`&`).

As the name implies, the *address operator* returns the address of its operand. This only works on operands that designate an object, so-called *lvalues*. For instance, the expression `&10` would be invalid since `10` does not designate an object, i.e. it has no associated address. The *indirection operator* performs the opposite operation, returning the designated object referenced by the

Expression	Terminal	Example
Basic pointer expressions		
Loading pointer symbols	`tpm_SymbolExp`	`p`
Indirection operator (dereferencing)	`tpm_UnaryExpDEREF`	`*p`
Address opeartor	`tpm_UnaryExpADDR`	`&a`
Pointer arithmetics		
Postincrement	`tpm_UnaryExpPOSTINC`	`p++`
Postdecrement	`tpm_UnaryExpPOSTDEC`	`p--`
Preincrement	`tpm_UnaryExpPREINC`	`++p`
Predecrement	`tpm_UnaryExpPREDEC`	`--p`
Addition	`tpm_BinaryExpPLUS`	`p + a`
Subtraction	`tpm_BinaryExpMINUS`	`p - a`
Subtraction (between pointers)	`tpm_BinaryExpMINUS`	`p - q`
Assignments to / with pointers		
Assigning a pointer	`tpm_AssignExpASSIGN`	`p = &a`
Addition and assignment	`tpm_AssignExpPLUS`	`p += a`
Subtraction and assignment	`tpm_AssignExpMINUS`	`p -= a`
Special expressions		
Arrow operator	`-` (handled specially)	`s->a`
Address and indirection operator	`tpm_UnaryExpDEREF/ADDR`	`&*a`

Table 6.1.: The various *pointer expressions* tackled in this thesis. `p` and `q` refer to *pointer symbols*, `a` is a symbol of *integer type* and `s` is a symbol *pointing to a composed type*.

given *pointer*. Therefore, the *indirection operator* always yields an *lvalue*. This operations is also called *dereferencing*.

Pointer arithmetics make up a big portion of the expressions tackled in this section. Generally, any *integer value* can be added on top of or subtracted from a pointer. Adding the value 1 to a pointer increases its address by the size of the type it is pointing to. For instance, adding the value 1 to an `int32_t*` would increase that pointer's value by 4 bytes, since `int32_t` is 4 bytes (32 bits) wide. Similarly, subtracting an *integer value* decreases that pointer's address in multiples of its base type's size.

The prefix and postfix increment and decrement operators (`++p`, `--p`, `p++`, `p--`) behave accordingly. The value 1 is added on top of or subtracted from the pointer, meaning the pointer's address is in- or decreased by the size of its base type. These operators, however, write back their result to the pointer `p`. Therefore, `p` itself has to be an *lvalue* for the expression to be valid. The *prefix* operator returns the pointer after the in- or decrement has been applied, whereas the postfix operator always returns the unmodified address of `p`.

Finally, C allows the programmer to subtract two pointers from one another. The result is an *integer value* that represents the number of elements that can be stored between the two pointers.

An ”element“ here referes to a value of the pointer's base type. Therefore, both pointers must point to the same base type for the subtraction to be a valid expression.

Similarly to the fact that pointer symbols have to be loaded, assigning new values to them also has to be considered. C offers shorthand syntaxes for arithmetics combined with assignments (`+=`, `-=`) which are taken into account here as well.

Finally, there are two use cases for pointers that deserve special attention. *Composed types* (`struct` and `union`) use the *component access operator* (`.`) to access their members. When working with pointers to `struct` or `union` types, it is often desired to *dereference* the pointer and then access one of its members. An expression combining both operators looks like this:

```
(*s).a
```

First, the `struct` pointer `s` is dereferenced and then its member `a` is accessed. Unfortunately, the parentheses are mandatory because the *component access operator* (`.`) has higher precedence than the *indirection operator* (`*`). To prevent the programmer from frequently having to use this clunky syntax, C offers a shorthand notation with the *arrow operator* (`->`). Writing `s->a` is then semantically equivalent to writing `(*s).a`.

The other special case is the chained use of *address* and *indirection operators*. The C standard mandates that the expression `&*p` shall generate the same code that would be output if only `p` was specified. Even if `p` was a nullpointer, the expression would not lead to undefined behavior.

6.2. Nonterminals

To realize a ruleset that is able to translate aforementioned expressions, several **nonterminals** have to be used. While some of them have been part of the ARMv4 *code selector* before, many have been added specifically for the purpose of translating expressions involving pointers. Table 6.2 presents the different relevant **nonterminals**.

The most basic **nonterminal** is `stmt`. It represents a statement in its entirety and is usually the *single, final nonterminal* left if tree pattern matching was successful.

A single register containing some kind of *value* is represented by the `reg` **nonterminal**. Since it is a *register* associated **action**-parts return a `LLIR_Register*`. Note again that, at this point, the vast majority of registers handled in the *code selector* are *virtual registers*. Usually, `reg` is used to store temporary values or represent local, non-stack-based symbols[1].

[1]The term *local, non-stack-based symbol* will be explained later in subsection 6.5.1.

Nonterminal	Return Type	Args.	Description
Nonterminals that were already part of the ARMv4 code selector			
`stmt`	`-`	`-`	Statement
`reg`	`LLIR_Register*`	`-`	Value register
`deref_reg`	`DerefInfo`	`bool`	Value register with memory location
`composed_type_object`	`address_with_offset`	`-`	`struct` or `union` object
Nonterminals new to the ARMv4 code selector that were added during this thesis			
`areg`	`LLIR_Register*`	`-`	Address register
`deref_areg`	`DerefInfo`	`bool`	Address register with memory location
`modified_areg`	`AddressModification`	`-`	Address register with pending modification
`zero_op_areg`	`LLIR_Register*`	`-`	Address register in a chain expression (`&*e`)
`zero_op_modified_areg`	`AddressModification`	`-`	Address register with pending modification in a chain expression (`&*e`)
`any_reg`	`LLIR_Register*`	`-`	Value / address register

Table 6.2.: **Nonterminals** used in the designed ruleset

The `deref_reg` **nonterminal** is used to represent a register holding a *value* that has a *memory location associated with it*. In other words, a `deref_reg` refers to a *designated object*, which precisely matches the definition of an *lvalue* presented in the previous section. The `DerefInfo` class holds all the relevant information and will be discussed in detail in section 6.4. The argument of type `bool` that has to be passed to its associated **action**-parts determines whether the result is *actually loaded*. If `false` is passed here, the `DerefInfo` class will only contain information on the object's *memory location*, but none whatsoever on its actual *value*. This will, for instance, be used when implementing the *address operator*, since the actual *value* of the object is not of interest there.

In a similar vein `areg` and `deref_areg` refer to registers containing an *address*. Treating *value* and *address* registers as two separate entities is a distinction only made in the *code selector*. In ARMv4 the same *physical registers* can be used to store and process *values* as well as *addresses*. Thereby the same types (`LLIR_Register*` and `DerefInfo`) are used to represent them. This distinction helps to easily differentiate rules implementing normal arithmetics from those implementing pointer arithmetics. For instance, a rule matching the `tpm_UnaryExpPOSTINC` **terminal** can specify whether its child node is a `reg` or an `areg`.

Note that the `areg` and `deref_areg` **nonterminals** were already part of the WCC's *internal code selector* for the *Infineon TriCore* processors, the target for which the WCC was originally developed. In that architecture, the distinction between a *data register* and an *address register* in the code selector has to be made because TriCore processors, unlike ARMv4, have a

heterogeneous register set. *Data* and *address registers* are physically distinct. [31]

Arguably the most important **nonterminal** of this thesis is `modified_areg` which is represented by the `AddressModification` class. This **nonterminal** represents an address register that has a *pending modification* associated with it. This means a modification to the *address register* has been scheduled, but no code has yet been inserted to realize the modification. The reason for postponing this modification will be explained in depth in subsection 6.3.2.

The `zero_op_areg` and `zero_op_modified_areg` **nonterminals** represent a single address register (with a *pending modification*) that is part of an expression with *chained address and indirection* operators (`&*e`). They simplify processing these special expressions. These two **nonterminals** have been modeled after a **nonterminal** with very similar name and identical purpose that was already part of the WCC's *TriCore code selector*. The `any_reg` **nonterminal** can represent both the `reg` and `areg` **nonterminal**. It is intended to be used in **tree** patterns where the rule does not differentiate between a *value* and an *address register*.

Finally, `composed_type_object` refers to objects of `struct` or `union` type. This **nonterminal** was defined by Paavo Becker as part of his bachelor thesis [10]. It is listed here since pointers to composed types have to be treated specially, which is why the rules presented in section 6.5 will refer to this **nonterminal** on some occasions. It is represented by the aptly named `address_with_offset` struct, which contains an address (`LLIR_Register*`) and a *byte offset*.

6.3. Auxiliary Class `AddressModification`

The `AddressModification` class is one of the two auxiliary classes that has been developed alongside the ruleset. It represents an *address register* that has a pending modification associated with it. The `modified_areg` **nonterminal** is represented by instances of this class.

The reason for this delay will become fully clear in subsection 6.3.2 where one of the class' member functions is explained in depth. The brief version is that the ARMv4 architecture allows for the combination of an address *modification* and a *memory access* to that address, potentially performing both within a single instruction. To utilize this architectural feature, rules matching an expression describing an *address modification* will not immediately emit corresponding instructions, but rather *postpone* them so that they can potentially be combined with a subsequent *memory access*. Emitting only a single instruction instead of multiple helps to cut down *code size* as well as *runtime*.

Figure 6.1 gives an overview of the class' *properties* as well as its three most important *public member functions*. At its core an object of type `AddressModification` describes an *address register*. A reference to this *address register* is either provided through a `LLIR_Register*` or through an object of type `DerefInfo`. The latter will be described in more depth in the

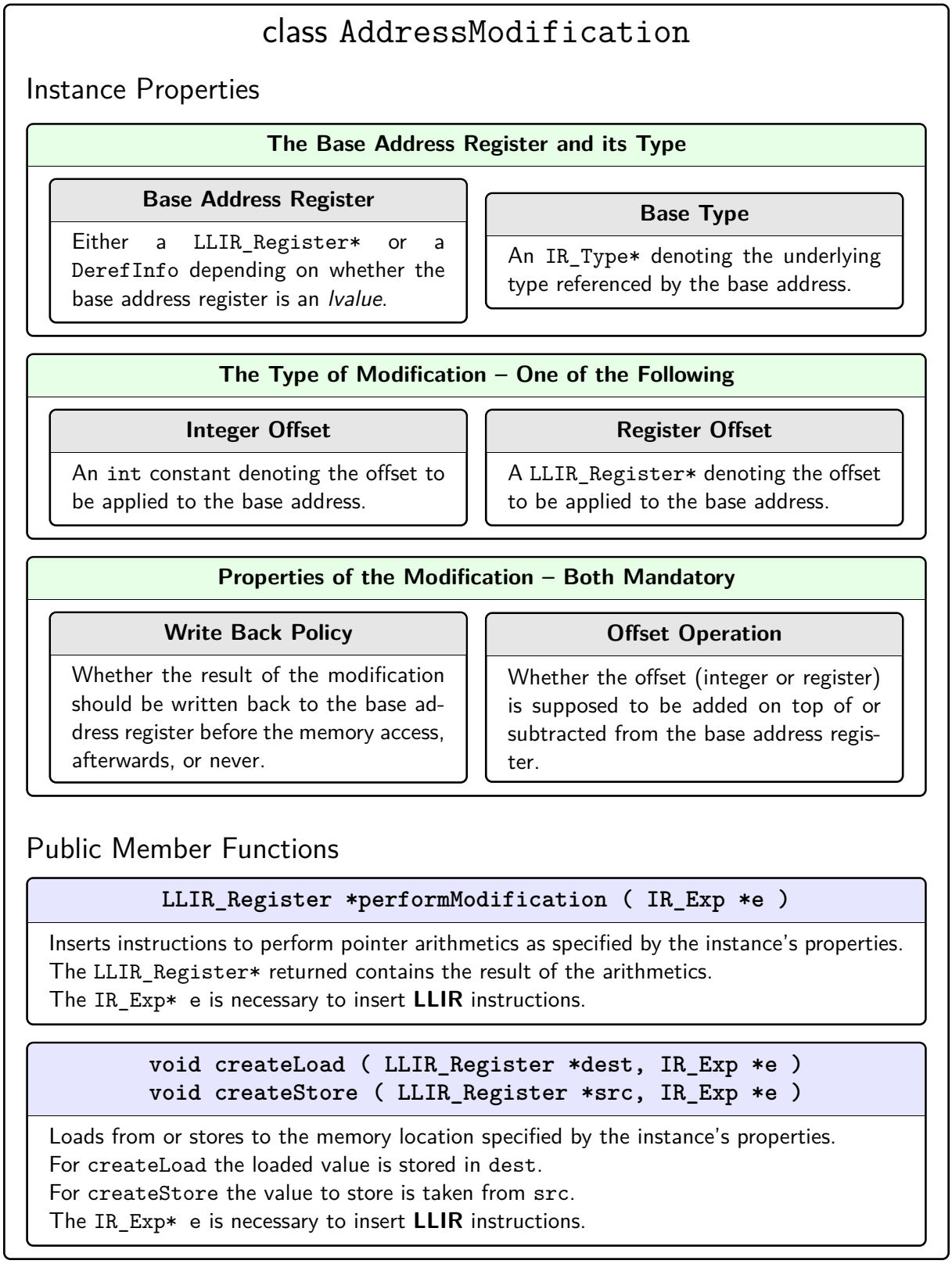


Figure 6.1.: Diagram describing key properties and methods of the `AddressModification` class

next section. Additionally, the *type* that the given *address register* points to is stored in the `AddressModification` object as well.

The other members are dedicated to the pending modification. Generally, there are two types of modifications that can be applied to an *address register*: One option is offsetting it by a *constant integer*. Note that this integer offset usually refers to *the number of base type elements* that the address should be offset with. For instance, if the *integer constant* 4 is given and the *base type* is 2 bytes wide, the actual *byte offset* that still has to be applied is 8. A flag can be set in the `AddressModification` class that allows for the specification of *raw byte offsets*, ignoring the address' *base type*.

The other option specifies an offset that is computed at runtime, residing in a `LLIR_Register*`. It, too, is assumed that the offset residing in the specified register refers to the *number of base type elements* that the base address should be offset with. Therefore, depending on the size of the *base type*, multiplication (or, if applicable, shifts) have to be performed on the *register offset* at runtime. The same flag mentioned in the previous paragraph can be used here too, causing the class to artificially set the size of the base type to 1 byte.

There are two properties of a modification that always have to be specified. The *write back policy* determines whether the result of the modification has to be written back to the *address register*. This is for instance the case in expressions like `p++` or `p+=a`. When *write back* is used, one can specify whether the value returned by the expression includes the modification (like in `++p`) or not (as in `p++`).

The other property specifies whether the offset should be added on top of or subtracted from the *base address*. This distinction applies to both *constant integer* as well as *register offsets*.

6.3.1. Public Member Function `performModification`

This public member function creates instructions to actually perform the modification as it is specified in the `AddressModification` instance. Tables 6.3 and 6.4 show the various instructions inserted for different *offset types*, *write back policies* and *offset operations* (addition or subtraction).

The $\texttt{p}_{\texttt{index}}$ symbol refers to a *register*. The following registers can be found in the given tables:

$\texttt{p}_{\texttt{base}}$	:	The *base address register* stored in the `AddressModification` instance.
$\texttt{p}_{\texttt{new}}$	:	A new *virtual register* allocated by `performModification`.
$\texttt{p}_{\texttt{return}}$	:	A new *virtual register* allocated by `performModification` for the special purpose of returning the old address value in *postfix write back* modifications.

Moreover, there are three different *immediate constants* used in both tables:

Write Back Policy	**Inserted Instructions**	**Returned Register**
Constant byte offset is a valid Addressing Mode 1 immediate		
None	`ADD/SUB` p_{new}, p_{base}, `#<offset>`	p_{new}
Prefix	`ADD/SUB` p_{base}, p_{base}, `#<offset>`	p_{base}
Postfix	`MOV` p_{return}, p_{base} `ADD/SUB` p_{base}, p_{base}, `#<offset>`	p_{return}
*Constant byte offset is **not** a valid Addressing Mode 1 immediate*		

None	`MOV` p_{new}, `#<offset`$_1$`>`
	`ORR` p_{new}, p_{new}, `#<offset`$_2$`>`
	`ORR` p_{new}, p_{new}, `#<offset`$_3$`>` (optional)
	`ORR` p_{new}, p_{new}, `#<offset`$_4$`>` (optional)
	`ADD/SUB` p_{new}, p_{base}, p_{new}
		p_{new}

Prefix	`MOV` p_{new}, `#<offset`$_1$`>`
	`ORR` p_{new}, p_{new}, `#<offset`$_2$`>`
	`ORR` p_{new}, p_{new}, `#<offset`$_3$`>` (optional)
	`ORR` p_{new}, p_{new}, `#<offset`$_4$`>` (optional)
	`ADD/SUB` p_{base}, p_{base}, p_{new}
		p_{base}

Postfix	`MOV` p_{new}, `#<offset`$_1$`>`
	`ORR` p_{new}, p_{new}, `#<offset`$_2$`>`
	`ORR` p_{new}, p_{new}, `#<offset`$_3$`>` (optional)
	`ORR` p_{new}, p_{new}, `#<offset`$_4$`>` (optional)
	`MOV` p_{return}, p_{base}
	`ADD/SUB` p_{base}, p_{base}, p_{new}
		p_{return}

Table 6.3.: Instructions generated by `performModification` for *constant integer offsets*

`#<offset>` : When a *constant integer offset* is used, this value refers to the final *byte offset* that has to be applied, with the *type size* already factored in.

`#<shift>` : If the size of the base type is a power of two, this value contains the basis value – in other words, it contains $\log_2(typesize)$.

`#<typesize>` : The size of the base type.

When talking about `performModification` the first case that has to be considered is not mentioned in the tables at all: The case where no modification is applied. One can build an `AddressModification` instance in which no modification is applied, boiling the class down to a variable containing an *address register*. If `performModification` is called on such instances, the function will simply return the *base address register* and generate no instructions.

The next group of cases, *constant integer offsets*, is detailed in table 6.3. The crucial distinction here is whether the *final byte offset* that has to be applied to the base address can be encoded as part of an *addition* [`ADD`] or *subtraction* [`SUB`]. Both of these instructions are *data-processing instructions* and adhere to *addressing mode 1* which has been detailed in subsection 3.2.1.

If the *final byte offset* is a valid *addressing mode 1* immediate, the modification itself can be performed with a single instruction. When no write back is used, the *byte offset* is simply added on top of or subtracted from the *base address*, with the result being stored in a new *virtual register*. The *offset operation* property of the instance determines whether addition or subtraction is performed.

When using the *prefix write back* policy the result is simply written over the *base address register* which is then also the *returned register* of the function. In *postfix write back* mode, the original *base address* is preserved by copying it into a new *virtual register* ($\texttt{p}_{\texttt{return}}$), which is then returned as the expression result by `performModification`. Afterwards, the modification is performed on the *base address* just as it would be when using *prefix write back*.

If the *final byte offset* cannot be encoded as an immediate operand in a *data-processing* instruction, it has to first be loaded. At least one *move* [`MOV`] and one *logical OR* [`ORR`] instruction have to be inserted to load the very big immediate offset. Depending on the nature of the offset, two more *logical OR* [`ORR`] instructions may have to be generated to fully load the constant byte offset.

Once the *byte offset* has been loaded, the procedure is essentially the same as it was before, with the `#<offset>` value replaced by the new virtual register $\texttt{p}_{\texttt{new}}$ holding the *final byte offset*.

Table 6.4 details the different instructions inserted when a *register offset* is used instead. *Constant integer offsets* can be multiplied with the *size of the base type* at compile-time, encoding the *final byte offset* in the generated instructions. When using *register offsets* this is not possible, since the size of the offset is not known at compile-time. Therefore, the multiplication of the offset with the *size of the base type* has to occur at runtime.

The crucial distinction made here is whether the *size of the base type* is a power of two. If it is, the multiplication boils down to a bit-shift, specifically a *logical shift left* (`lsl`), which is a much simpler operation and can usually be performed faster than full-blown integer multiplication. There is another catch to it, however: The *shifter operands* discussed in subsection 3.2.1 allow us to specify immediate shifts *as part of a data-processing operand*. Therefore, no separate *shift instruction* has to be inserted at all.

The first section of table 6.4 details the instructions inserted if the size of the base type is indeed a power of two. The instructions themselves are very similar to those presented in table 6.3 that were discussed previously. However, the immediate constant `#<offset>` has been replaced with the *register offset* $\texttt{p}_{\texttt{offset}}$ which is shifted left by the appropriate amount of bits `#<shift>` before being added to or subtracted from the *base address register* $\texttt{p}_{\texttt{base}}$.

If the size of the base type is **not** a power of two, it has to be loaded first. In most cases this will boil down to a simple *move* [`MOV`] instruction loading the type size. In some obscure cases, however, the size of the base type may be too big for it to be encoded as a single *immediate*

Write Back Policy	**Inserted Instructions**	**Returned Register**
	Size of base type is a power of two	
None	`ADD/SUB` p_{new}, p_{base}, p_{offset}, `lsl #<shift>`	p_{new}
Prefix	`ADD/SUB` p_{base}, p_{base}, p_{offset}, `lsl #<shift>`	p_{base}
Postfix	`MOV` p_{ret}, p_{base} `ADD/SUB` p_{base}, p_{base}, p_{offset}, `lsl #<shift>`	p_{ret}
	*Size of base type is **not** a power of two*	
None	`MOV` p_{new}, `#<typesize`$_1$`>` `ORR` p_{new}, `#<typesize`$_2$`>` (optional) `ORR` p_{new}, `#<typesize`$_3$`>` (optional) `ORR` p_{new}, `#<typesize`$_4$`>` (optional) `MUL` p_{new}, p_{new}, p_{offset} `ADD/SUB` p_{new}, p_{base}, p_{new}	p_{new}
Prefix	`MOV` p_{new}, `#<typesize`$_1$`>` `ORR` p_{new}, `#<typesize`$_2$`>` (optional) `ORR` p_{new}, `#<typesize`$_3$`>` (optional) `ORR` p_{new}, `#<typesize`$_4$`>` (optional) `MUL` p_{new}, p_{new}, p_{offset} `ADD/SUB` p_{base}, p_{base}, p_{new}	p_{base}
Postfix	`MOV` p_{new}, `#<typesize`$_1$`>` `ORR` p_{new}, `#<typesize`$_2$`>` (optional) `ORR` p_{new}, `#<typesize`$_3$`>` (optional) `ORR` p_{new}, `#<typesize`$_4$`>` (optional) `MUL` p_{new}, p_{new}, p_{offset} `MOV` p_{return}, p_{base} `ADD/SUB` p_{base}, p_{base}, p_{new}	p_{return}

Table 6.4.: Instructions generated by `performModification` for *register offsets*

constant in *addressing mode 1.* In those cases, up to three additional *logical OR* [`ORR`] instructions may be inserted to load the full type size.

After the type size has been loaded a *multiplication* [`MUL`] is inserted, multiplying the just loaded *type size* with the instance's *register offset.* The actual *addition* [`ADD`] or *subtraction* [`SUB`] instruction performing the modification behaves very similarly to those previously discussed and will not be explained further here. Note that the *multiplication* [`MUL`] instruction does not allow the specification of *immediate operands*, making it necessary to load the *type's size* with the seperate *move* [`MOV`] (and *logical OR* [`ORR`]) instructions.

Finally, there is one aspect to the `performModification` function that has been omitted so far. If the *base address register* is not stored in the *AddressModification* instance as a `LLIR_Register*` but instead as a `DerefInfo`, special attention has to be put towards the *postfix and prefix write back policies.*

Simply overwriting the `LLIR_Register*` that holds the value of the *base address* does not suffice when said *base address* is an *lvalue*. In those cases, the updated *base address* has to be written back to the *designated memory location* as well. Storing the updated *base address* is performed with the `DerefInfo`'s public member function `storeBack`, which will be discussed in section 6.4. This code has intentionally been left out of tables 6.3 and 6.4 to keep the presented information focused on the task of *address modification*.

6.3.2. Public Member Functions `createLoad` and `createStore`

The `createLoad` and `createStore` member functions of the `AddressModification` class are essentially the reason for its existence. These two member functions are the whole reason for postponing pointer arithmetics in the first place. `createLoad` will insert all the necessary instructions to load a value from the memory location that is specified by the `AddressModification` instance. Similarly, `createStore` will store a value into the specified location.

One may now interject that the `AddressModification` class does not, in fact, specify a simple *memory location*, but rather a *memory address* with a *pending modification* attached to it. Thanks to *addressing modes 2 and 3* discussed in subsection 3.2.2, both *the modification* and *the memory access* can potentially be combined into a single instruction. This feature of the ARMv4 architecture will be made use of across the entire *pointer expression* ruleset. The extent of it will become clear when discussing the ruleset in section 6.5.

Tables 6.5 and 6.6 provide an overview of the different instructions generated for *constant integer* and *register offsets* respectively. The referenced *registers* and *immediate constants* are the same ones used in tables 6.3 and 6.4 and will not be discussed again. The only new addition is $\texttt{p}_{\texttt{src/dst}}$, which refers to the *destination register* during a load or the *source register* during a store. As shown in figure 6.1, the `LLIR_Register*` of this register is passed to `createLoad` / `createStore` as an argument.

First up is again the scenario that the `AddressModification` instance does not describe any modification at all, boiling down to a container for an *address register*. In those cases, the generated instruction is akin to specifying a *constant integer offset* of 0:

$$\texttt{LDRXX/STRXX } \texttt{p}_{\texttt{src/dst}}\texttt{, [p}_{\texttt{base}}\texttt{, \#0]}$$

For `createLoad` `LDRXX` is used, whereas for `createStore` `STRXX` is inserted. Note that `XX` is a placeholder here, referring to the many different *load and store variants* available in the ARMv4 architecture, which were discussed in subsection 3.2.2 and presented in table 3.2. The variant is chosen based on the *base type* specified in the `AddressModification` instance.

The instructions inserted for *constant integer offsets* are presented in table 6.5. Similarly to `performModification`, the differentiation has to be made whether the *final byte offset* can be

Write Back Policy	**Inserted Instructions**
Constant byte offset is a valid Addressing Mode 2/3 immediate	
None	`LDRXX/STRXX` $p_{src/dst}$`, [`p_{base}`, #±<offset>]`
Prefix	`LDRXX/STRXX` $p_{src/dst}$`, [`p_{base}`, #±<offset>]!`
Postfix	`LDRXX/STRXX` $p_{src/dst}$`, [`p_{base}`], #±<offset>`
*Constant byte offset is **not** a valid Addressing Mode 2/3 immediate*	

None	`MOV` p_{new}`, #<`$offset_1$`>`
	`ORR` p_{new}`, #<`$offset_2$`>` (optional)
	`ORR` p_{new}`, #<`$offset_3$`>` (optional)
	`ORR` p_{new}`, #<`$offset_4$`>` (optional)
	`LDRXX/STRXX` $p_{src/dst}$`, [`p_{base}`, ±`p_{new}`]`

Prefix	`MOV` p_{new}`, #<`$offset_1$`>`
	`ORR` p_{new}`, #<`$offset_2$`>` (optional)
	`ORR` p_{new}`, #<`$offset_3$`>` (optional)
	`ORR` p_{new}`, #<`$offset_4$`>` (optional)
	`LDRXX/STRXX` $p_{src/dst}$`, [`p_{base}`, ±`p_{new}`]!`

Postfix	`MOV` p_{new}`, #<`$offset_1$`>` `ORR` p_{new}`, #<`$offset_2$`>` (optional) `ORR` p_{new}`, #<`$offset_3$`>` (optional) `ORR` p_{new}`, #<`$offset_4$`>` (optional) `LDRXX/STRXX` $p_{src/dst}$`, [`p_{base}`], ±`p_{new}

Table 6.5.: Instructions generated by `createLoad`/`createStore` for *constant integer offsets*

encoded as an immediate operand in the instruction. Thanks to the variety of *load and store variants* available, two different *addressing modes* have to be taken into account. As was stated in subsection 3.2.2, *addressing mode 2* allows for immediate operands that are up to 12 bits wide, whereas *addressing mode 3* only allows for 8 bits. Which mode is applicable depends on the chosen *load and store variant*, which in turn depends on the *base type* specified in the `AddressModification` instance.

If possible, the offsets are then encoded as part of the instruction. If a *postfix or prefix write back policy* was specified, the *pre-indexing* or *post-indexing* load and store variants are used. These variants update the value of the *base address register* and were presented in table 3.3.

If the *final byte offset* cannot be encoded as part of the instruction, it is loaded manually with a *move* [`MOV`] and optionally up to three *logical OR* [`ORR`] instructions. The actual load or store instruction then uses the p_{new} register in which the offset was previously loaded. The *pre-* and *post-indexing* modes are used here too, preventing the insertion of an *addition* [`ADD`] or *subtraction* [`SUB`] that would update the *base address register*.

Table 6.6 presents the various snippets of code inserted for *register offsets*. For register offsets the multiplication with the *size of the base type* has to occur at runtime. Therefore, once again,

Addressing Mode	**Write Back Policy**	**Inserted Instructions**
	Size of base type is a power of 2	
Mode 2	None	`LDRXX/STRXX` $p_{src/dst}$`, [`p_{base}`,` $\pm p_{offset}$`, lsl #shift]`
	Prefix	`LDRXX/STRXX` $p_{src/dst}$`, [`p_{base}`,` $\pm p_{offset}$`, lsl #shift]!`
	Postfix	`LDRXX/STRXX` $p_{src/dst}$`, [`p_{base}`],` $\pm p_{offset}$`, lsl #shift`
Mode 3	None	`MOV` p_{new}`,` p_{offset}`, lsl #shift` `LDRXX/STRXX` $p_{src/dst}$`, [`p_{base}`,` $\pm p_{new}$`]`
	Prefix	`MOV` p_{new}`,` p_{offset}`, lsl #shift` `LDRXX/STRXX` $p_{src/dst}$`, [`p_{base}`,` $\pm p_{new}$`]!`
	Postfix	`MOV` p_{new}`,` p_{offset}`, lsl #shift` `LDRXX/STRXX` $p_{src/dst}$`, [`p_{base}`],` $\pm p_{new}$
	*Size of base type is **not** a power of 2*	

Modes 2 & 3	None	`MOV` p_{new}`, #<typesize`$_1$`>`
		`ORR` p_{new}`, #<typesize`$_2$`>` (optional)
		`ORR` p_{new}`, #<typesize`$_3$`>` (optional)
		`ORR` p_{new}`, #<typesize`$_4$`>` (optional)
		`MUL` p_{new}`,` p_{new}`,` p_{offset}
		`LDRXX/STRXX` $p_{src/dst}$`, [`p_{base}`,` $\pm p_{new}$`]`

	Prefix	`MOV` p_{new}`, #<typesize`$_1$`>`
		`ORR` p_{new}`, #<typesize`$_2$`>` (optional)
		`ORR` p_{new}`, #<typesize`$_3$`>` (optional)
		`ORR` p_{new}`, #<typesize`$_4$`>` (optional)
		`MUL` p_{new}`,` p_{new}`,` p_{offset}
		`LDRXX/STRXX` $p_{src/dst}$`, [`p_{base}`,` $\pm p_{new}$`]!`

	Postfix	`MOV` p_{new}`, #<typesize`$_1$`>` `ORR` p_{new}`, #<typesize`$_2$`>` (optional) `ORR` p_{new}`, #<typesize`$_3$`>` (optional) `ORR` p_{new}`, #<typesize`$_4$`>` (optional) `MUL` p_{new}`,` p_{new}`,` p_{offset} `LDRXX/STRXX` $p_{src/dst}$`, [`p_{base}`],` $\pm p_{new}$

Table 6.6.: Instructions generated by `createLoad`/`createStore` for *register offsets*

the important distinction is whether the *size of the base type* is a power of two, since then the multiplication boils down to a *logical shift left*.

If the *size of the base type* is indeed a power of two and the used *load or store variant* uses *addressing mode 2*, modification and memory access can be encoded in a single instruction. The *register offset* `p_offset` is first shifted left by the appropriate number of bits `#<shift>`, before being added on top of `p_base`. Once again, if a *prefix or postfix write back policy* was specified, *pre-* and *post-indexing* load or store variants are used to realize the write back.

Unfortunately, *addressing mode 3* does not support *shifted register offsets*. For load and store variants conforming to this addressing mode, the offset has to first be shifted manually by inserting a *move* [`MOV`] instruction that applies the appropriate shift to the *register offset*. When the *size of the base type* is a power of two, this approach is still preferable over using full-blown integer *multiplication* [`MUL`], since a single shift can potentially be executed much faster.

If the *size of the base type* is not a power of two, an approach very similar to the one seen for `performModification` has to be used. The *size of the base type* is first loaded into `p_new`, then an instruction for multiplication at runtime is inserted and finally, the calculated *byte offset* residing in `p_new` is applied to the base register. As has been the case before, the *write back policies* correspond to the load and store indexing modes here as well.

Additionally, note that the same omission has been made in tables 6.5 and 6.6 that was made in tables 6.3 and 6.4. If the *base address register* is an *lvalue*, meaning it is represented by an instance of class `DerefInfo`, and a *prefix or postfix write back policy* has been specified, the updated value of the *base address register* has to be written back to its *designated memory location*. This is realized using the public member function `storeBack` of class `DerefInfo`.

Finally, there is one special feature of the `createLoad` / `createStore` pair that has not been discussed yet. Inserting all the auxiliary instructions that, for instance, load the type size or multiply the *register offset* with the *type size* at runtime is expensive. In the event that both a *load* and a *store* has to be inserted, a special property named `mPrecomputedOffset` of type `LLIR_Register*` stores a reference to register `p_new`. There is also a boolean flag, `mModApplied`, that tracks whether a call to `createLoad` or `createStore` has already been made. If, for instance, `createStore` is called after `createLoad` had already been invoked on the same `AddressModification` instance, the expensive instructions mentioned previously would not be inserted again. Instead, `p_new` which already contains the processed *final byte offset* will be reused. This approach guarantees that the call to `createStore` will then at most only ever insert a single *store variant* [`STRXX`] into the **LLIR**.

6.4. Auxiliary Class `DerefInfo`

class `DerefInfo`

Instance Properties

`LLIR_Register *mResultRegister`

An *lvalue* is a *value* that has a *memory location* associated with it.
`mResultRegister` contains the *value* of this instance. Can be a `nullptr`, in the event that the *value* itself has not been loaded.

`AddressModification mAddress`

An *lvalue* is a *value* that has a *memory location* associated with it.
`mAddress` refers to the *memory location* of this instance. Due to the nature of the `AddressModification` class, there may be a pending modification associated with it.

`IR_Type *mDataType`

The *type* of `mResultRegister`, which is the *base type* of `mAddress`.

Public Member Functions

`LLIR_Register *calculateAddress( IR_Exp *e )`

Calls `mAddress.performModification( e )` and returns its result.
If a modification was still pending, the necessary instructions will be inserted now.

`void storeBack( LLIR_Register *source, IR_Exp *e )`

Calls `mAddress.createStore( source, e )`.
Can be used to store a new *value* into the *lvalue* represented by this instance.

Figure 6.2.: Diagram describing key properties and methods of the `DerefInfo` class

The `DerefInfo` class is used to represent *lvalues*, *values* that have a *designated memory location* associated with them. Figure 6.2 presents the class' most important *properties* and *public member functions*. Note that this class already existed as a `struct` type before work on this thesis began. It has, however, been completely rewritten to integrate it with the new `AddressModification` class.

The purpose of this class is best illustrated with a small example. Consider a local, stack-based variable[2] that was initialized with the expression `int foo = 5;`. This variable resides somewhere on the stack, and its precise address is this object's *designated memory location*. However, `foo` also represents a simple *value*, in this case `5`.

The `mResultRegister` property is a register (`LLIR_Register*`) that contains the *value* of this

[2]The term *local, stack-based variable* will be discussed in depth in subsection 6.5.1.

`DerefInfo`. Note that this property does not have to be set, i.e. `mResultRegister` can be a `nullptr`. In those cases the actual *value* of the *lvalue* is not of interest and was therefore never loaded. Instead, the instance's purpose is solely to refer to a *designated memory location*.

The `mAddress` member is of type `AddressModification` which was just discussed in section 6.3. It stores the object's *memory location*. Due to the nature of the `AddressModification` class this memory location may still have a *pending modification* associated with it. Nonetheless, thanks to the `AddressModification`'s public member functions, values can be loaded from or stored into this location as if it were a simple address. This property should always be defined, otherwise the instance would not represent an *lvalue*.

Finally, the `DerefInfo` also contains the type (`IR_Type*`) of the *lvalue*. This property is here for completeness' sake since the `mAddress` property stores the *base type* of its address anyways.

The `DerefInfo`'s two most important member functions are essentially just gateways to the previously discussed member functions of the `AddressModification` class.

The `calculateAddress` method calls `performModification` on `mAddress` and returns its result. This function should be used when the `DerefInfo`'s value is not important, but its actual address must be extracted. The `storeBack` method calls `createStore` on `mAddress`, passing through the `LLIR_Register*` of the source value. This is the same method that was already referenced in the previous section, for the cases where a *prefix or postfix write back policy* forced the `AddressMoficiation` class to write back the updated *base address register* to its memory location.

Note that no method exists in the `DerefInfo` class to load from its memory location. It is assumed that the value is loaded when constructing the class instance, with the register containing the result being referenced through `mResultRegister`.

6.5. OLIVE Rules

This subsection will finally dive into the actual ruleset used to translate pointer expressions. Since a total of 42 different rules will be discussed here, with their combined source code well exceeding a thousand lines, individual code snippets will not be examined in this section.

Instead, semantically related rules will be grouped together and presented as a diagram box as can be seen in figure 6.3. Every diagram lists the brief purpose of the rule(s), the **nonterminal(s)** they represent, the *arguments* of their associated **action**-parts and the **tree** pattern they match. **Action**- and **cost**-part of these rules have been omitted in the diagram, since these will be discussed in text.

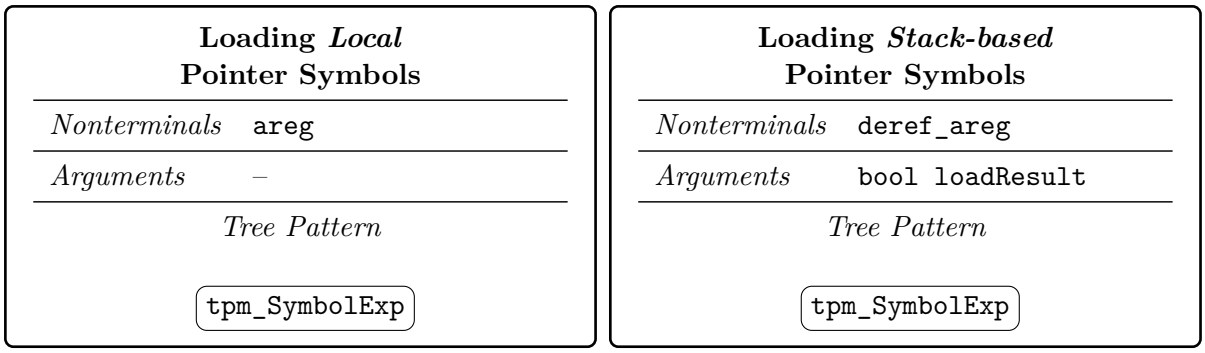


Figure 6.3.: Rules that load pointer symbols

Moreover, note that a single diagram box can represent multiple rules. When a diagram lists multiple **nonterminals** or multiple options for **tree nodes**, it represents every possible combination of options, with each combination implemented as a single rule. For instance, figure 6.8 shows the rule for addition or subtraction in the context of pointer arithmetics. Two options are shown for the parent terminal (`tpm_BinaryExpPLUS` and `tpm_BinaryExpMINUS`) and for the right child node (`reg` and `tpm_IntConstExp`). This means the actual ruleset implements $2 \cdot 2 = 4$ different rules, one for each possible combination.

6.5.1. Loading Pointer Symbols

Figure 6.3 presents the first two rules that will be discussed. Both match against the `tpm_SymbolExp` **terminal**, which represents a symbol being present in an expression. A rule like this already existed for the `reg` **nonterminal**, but still had to be implemented for the `areg` and `deref_areg` **nonterminals**.

When observing these two rules, one might be inclined to ask how a symbol can be represented by the `reg` or `areg` **nonterminals** in the first place. Any symbol defined in C, be it a simple local variable or a global or static symbol has a *memory location* associated with it. This is the reason why the *address operator* (`&`) can be used on these symbols.

The WCC employs a little trick here by distinguishing between *local register symbols* and *local stack symbols*. It automatically classifies every symbol that *does not have to be represented in memory* as a *local register symbol*. For instance, a local loop counter that only ever serves the purpose of managing the loop count does not have to reside in memory. Constantly reading its value from memory and writing its increment back to memory is incredibly expensive, and doing so delivers no benefit whatsoever. Instead, it would be a lot more efficient to simply keep this symbol in the *register space* for its entire lifetime.

And that is exactly what *local register symbols* are: Symbols that reside exclusively in the register space for their entire lifetime and therefore have no memory address associated with them. Should the amount of *local register symbols* exceed the number of available *physical registers* on

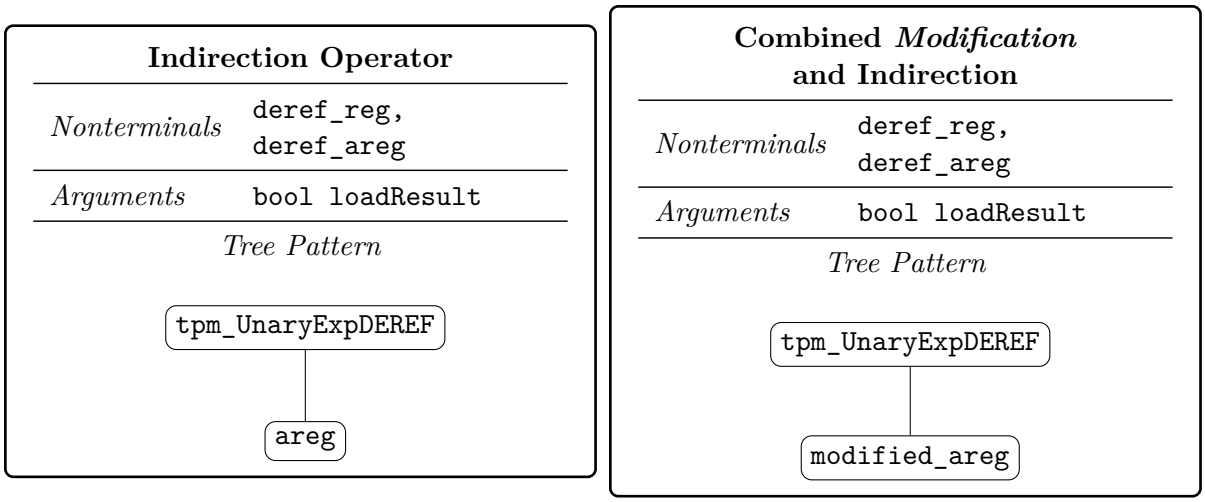


Figure 6.4.: Rules for the *indirection operator* (`*`)

the processor, the *register allocator* will create so-called *spill-code*, temporarily offloading their contents on to the stack.

The *code selector* framework already offers functions that allow us to determine whether a symbol is a *local register* or a *local stack symbol*. Moreover, it offers functions that look up these symbols in the symbol table and create the necessary instructions to load them for us. Therefore, the rules themselves simply invoke these already existing functions.

The cost, too, is calculated by functions already provided by the *code selector* framework. Note, that if the rule loading a *local register symbol* attempts to load a *local stack symbol* or vice versa, the **cost** of that rule will be *infinity*. Whenever a rule's **cost** evaluates to *infinity*, the **ICD-CG** will never use it for matching.

One special thing to note about the rule representing the `deref_reg` **nonterminal** is the *argument* to its **action**-part. The `loadResult` argument determines whether the *value* of the symbol should actually be loaded. If `false` is provided here, only the *memory location* will be stored in the `DerefInfo` and `mResultRegister` will remain a `nullptr`.

6.5.2. Indirection Operator

Figure 6.4 shows the two most important rules when it comes to matching the *indirection operator* (`*`). The rule on the left shows the traditional implementation matching against the *indirection operator* itself (`tpm_UnaryExpDEREF`) which has a simple *address register* `areg` as its child node.

The rule shown on the right side of figure 6.4 shows the advanced implementation which has been hinted at in previous subsections. Thanks to the `AddressModification` class and its `createStore` / `createLoad` member functions, *memory accesses* and *modifications* can possibly

be combined into a single instruction. This rule enables exactly that, by taking an *address register with a pending modification* (`modified_areg`) as the child node to the *indirection operator* (`*`).

Both rules first call the **action**-part of its child-node. The rule taking an `areg` as its child node then packages the *address register* in a new `AddreddModification` instance with no offset. Both rules then proceed to package the `AddressModification` instance in a `DerefInfo`, optionally calling `createLoad` on the `AddressModification` object if `loadResult` was set to `true`.

The **cost**-parts of both rules give *worst-case* estimates of the actual costs. Unfortunately, it is impossible to determine the value of the argument `loadResult` in the **cost**-part. Moreover, only the **action**-part of the rule taking the `modified_areg` can actually interact with the `AddressModification` object. The **cost**-part is not given any indication as to what type of modification is given in the `AddressModification`.

The **cost**-parts of both rules assume that `loadResult` is always true. The rule taking the `areg` **nonterminal** gives a more accurate cost estimation. It creates an `AddressModification` object with the correct properties, mainly the *base type* and the *type of modification* (none). Then, a previously undiscussed public member function is called, `createLoadCost`. This function is structured almost identically as `createLoad`, but instead of actually inserting the respective instructions, it accumulates their respective costs and returns the sum.

The rule taking the `modified_areg` **nonterminal** gives an imprecise cost estimation. They call a *static* member function of class `AddressModification` which returns the absolute worst cost that could be incurred when calling `createLoad` on a real instance. That cost consists of loading a very big type size (one *move* [`MOV`] and three *logical OR* [`ORR`] instructions), *multiplying* [`MUL`] at runtime, *loading* [`LDR`] the value from memory and finally having to insert a *store* [`STR`] to update the *base address* in case it is an *lvalue* and a *postfix or prefix write back policy* was specified.

6.5.3. Address Operator

The counterpart of the *indirection operator*, the *address operator* (`&`), is matched by the rules shown in figure 6.5. Both rules match the **terminal** representing the *address operator* itself.

The rule on the left of the figure matches registers that represent *values* as well as *addresses*. However, both must have a *memory location* associated with them, otherwise no *address* can be evaluated. Therefore, the rule on the left matches the `deref_reg` and `deref_areg` **nonterminals**, and not the `reg` and `areg` **nonterminals**.

The **action**-part evaluates the **action**-part of its child node, setting the value of argument `loadResult` to `false`, since only the *memory location* is of interest to the *address operator*. It then calls `calculateAddress` on the returned `DerefInfo` instance to retrieve a `LLIR_Register*`

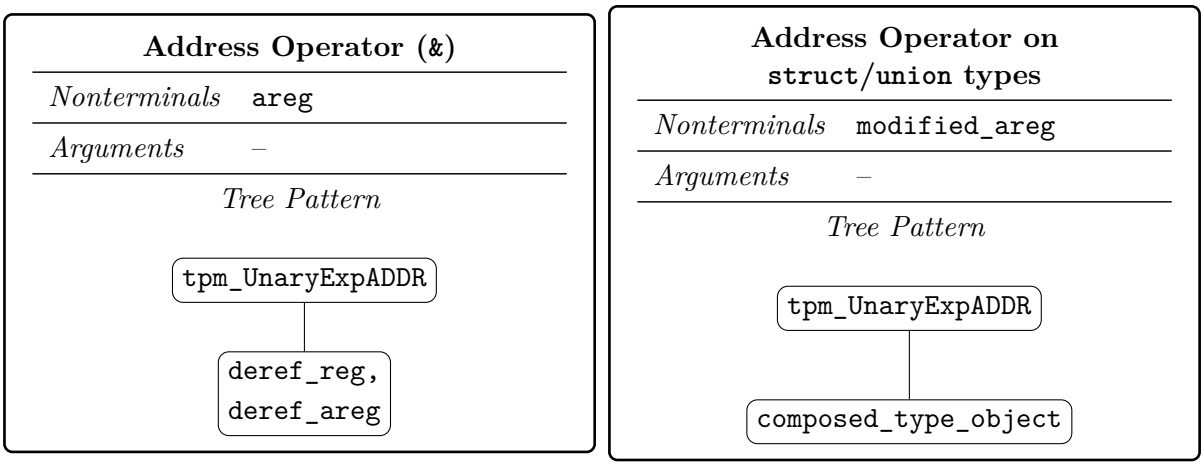


Figure 6.5.: Rules matching the *address operator* (`&`)

containing the actual address. If the `DerefInfo`'s `mAddress` member still contained any *pending modifications*, they would be applied now. Please refer to subsection 6.3.1 for details on `performModification`'s behavior, which is the method invoked by `calculateAddress`.

The **cost**-part consists of the cost of the child-node as well as another worst-case estimation of the cost it would take to extract the address out of the `DerefInfo`. Ultimately, this estimation boils down to the worst case estimation for `performModification`, which consists of a *move* [`MOV`] because of the *postfix write back policy*, another *move* [`MOV`] and three *logical OR* [`ORR`] instructions for loading a very big type size, a *multiplication* [`MUL`] at runtime, the actual modification (*addition* [`ADD`]) and finally a *store* [`STR`] to write back the updated *base address* in case it is an *lvalue*.

The rule presented on the right side of figure 6.5 works a little differently since it processes a `composed_type_object` **nonterminal**. As was briefly mentioned in section 6.2, the `composed_type_object` **nonterminal** is represented by the `address_with_offset` struct. It contains an address (`LLIR_Register*`) with a *byte offset*. It is important to understand that `struct` and `union` objects translated by the WCC's internal ARMv4 code selector are always stored in memory and their symbols therefore consist of *an address pointing to the location* where the actual object resides. Local symbols use the *stack pointer* [`sp`] with a byte offset, while global symbols use references to the respective section in the program's memory, which are later replaced with the actual addresses by the *assembler* and *linker*.

In other words, composed types *already use pointers*. To determine the address of a composed type, no actual work needs to be performed and no instructions need to be inserted. The `LLIR_Register*` and *byte offset* of the `address_with_offset` struct are simply repackaged into an object of type `AddressModification`. That is why the rule returns a `modified_areg` **nonterminal** and why its **cost**-part only consists of the cost of the `composed_type_object` child-node.

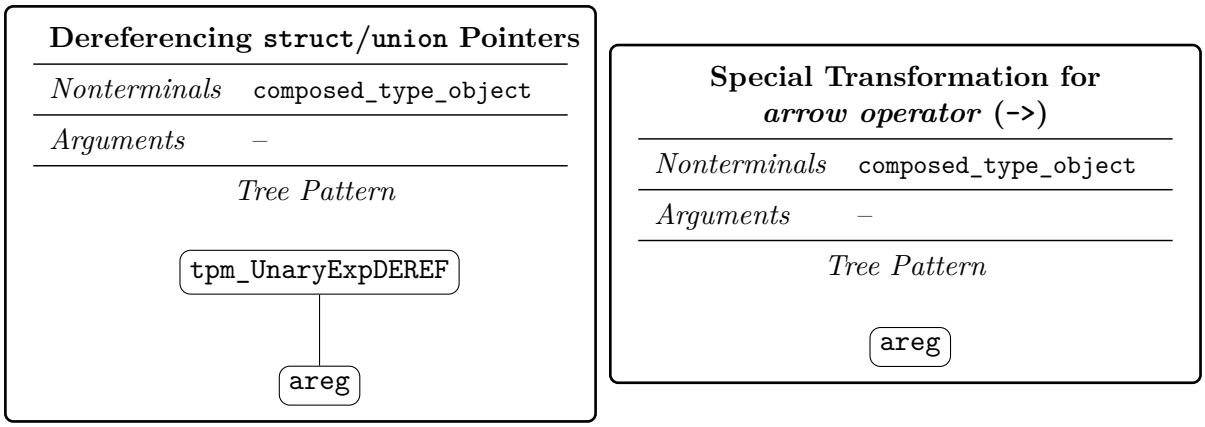


Figure 6.6.: Special rules for dereferencing composed types

6.5.4. Indirection and Component Access for Composed Types

Figure 6.6 presents two special rules that implement *indirection* for composed types. On the left side of the figure, the rule handling the dereferencing of a pointer to a composed type (`struct` or `union`) is presented. The rule on the right side of the figure handles the special case of combined *indirection and component access*, the *arrow operator* (`->`).

When applying the *address operator* to a composed type, no actual instructions had to be inserted since `struct` and `union` objects are already handled as *addresses* internally. The opposite is therefore true as well: When dereferencing a pointer, no actual instructions have to be inserted. Instead, the **action**-part simply repackages the *address register* matched as the child-node (`areg`) into the `address_with_offset` struct. Similarly, the **cost**-part simply returns the cost of the matched child-node. The only exception occurs when the child `areg` does not point to a composed type, in which case the **cost** of the rule is *infinity*.

The rule matching the *arrow operator* is implemented in an unusual fashion. The problem is that both forms of *component access operators*, the *dot* (`.`) as well as the *arrow operator* (`->`), are represented by the same **terminal**, `tpm_ComponentAccessExp`. The *dot operator* was already implemented by Paavo Becker as part of his bachelor thesis on composed types [10].

Practically, there is little difference between the two operators, since `composed_type_objects` are internally already handled as *pointers*. The rule presented on the right of figure 6.6 transforms an `areg` **nonterminal** pointing to a composed type to the `composed_type_object` **nonterminal**. The **action**-part simply repackages the address into the `address_with_offset` struct and no instructions are generated.

The core of this rule lies within its **cost**-part. The code uses the **ICD-C** interface to check whether the expression represented by the `areg` nonterminal indeed refers to a composed type and whether the expression surrounding the `areg` **nonterminal** is a *component access expression* using the *arrow operator*. If all conditions are met, the rule can be applied and the actual **cost**

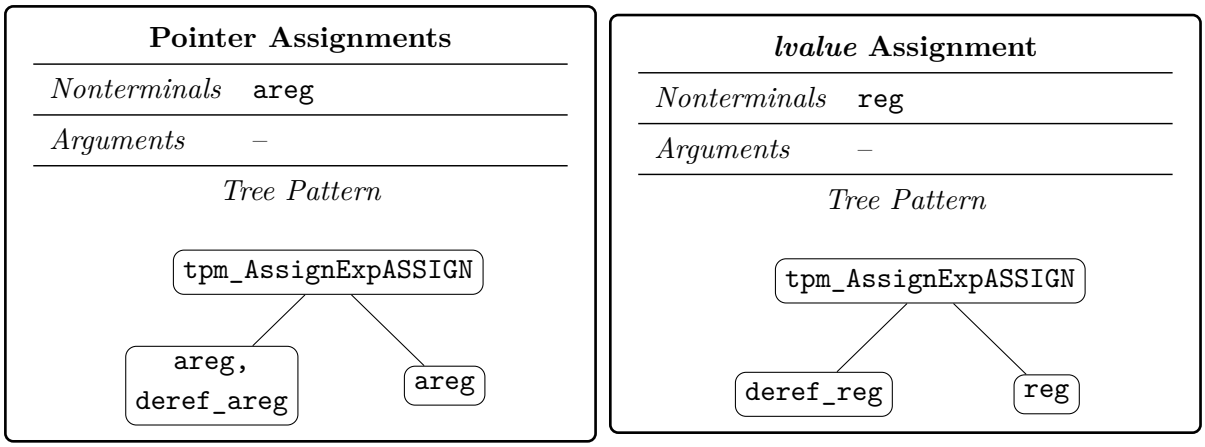


Figure 6.7.: Rules matching *assignment operators* (=)

amounts to that of the child-node. If not, the rule's cost will evaluate to *infinity*, and the **ICD-CG** will not consider it for matching.

6.5.5. Pointer and lvalue Assignments

The rules presented in figure 6.7 all perform *assignments* using the *assignment operator* (=), which is represented by the `tpm_AssignExpASSIGN` **terminal**. Since the `DerefInfo` class has been completely redone, its assignment rule too has been reimplemented, represented by the rule labeled *lvalue Assignment*.

The approach for all assignments is essentially the same. The **action**-part first evaluates the *right-hand side* of the assignment. Then, the *left-hand side* of the assignment is evaluated. For the rules that match the `deref_reg` or `deref_areg` **nonterminals** as their *left-hand side*, their **action**-parts are invoked with `loadResult` set to false. Only the *lvalue*'s memory location is important, its old value does not have to be acquired.

For the rules matching the *lvalues* as their left-hand side, the `DerefInfo`'s `createStore` member function is now called, with the `LLIR_Register*` of the *right-hand side* passed as the `source` argument. If the *left-hand side* is a *local register symbol* a simple *move* [`MOV`] is inserted instead, copying the contents of the *right-hand side* to the *left-hand side*.

The cost of all three rules always includes the cost of its child-nodes. For the rule matching the `areg` **nonterminal** as its *left hand side* the cost of the inserted *move* [`MOV`] is added. Rules matching *lvalues* as their *left hand side* add an estimation of the worst-case cost when calling `createStore`. Interestingly, this estimation only consists of a single instruction, a *store* [`STR`]. It is assumed that the worst-case estimate for `createLoad` has already been factored into the tree cost when creating the `deref_reg` or `deref_areg` **nonterminal**. If one would then call `createStore` afterwards, thanks to the `AddressModification`'s `mPrecomputedOffset` property

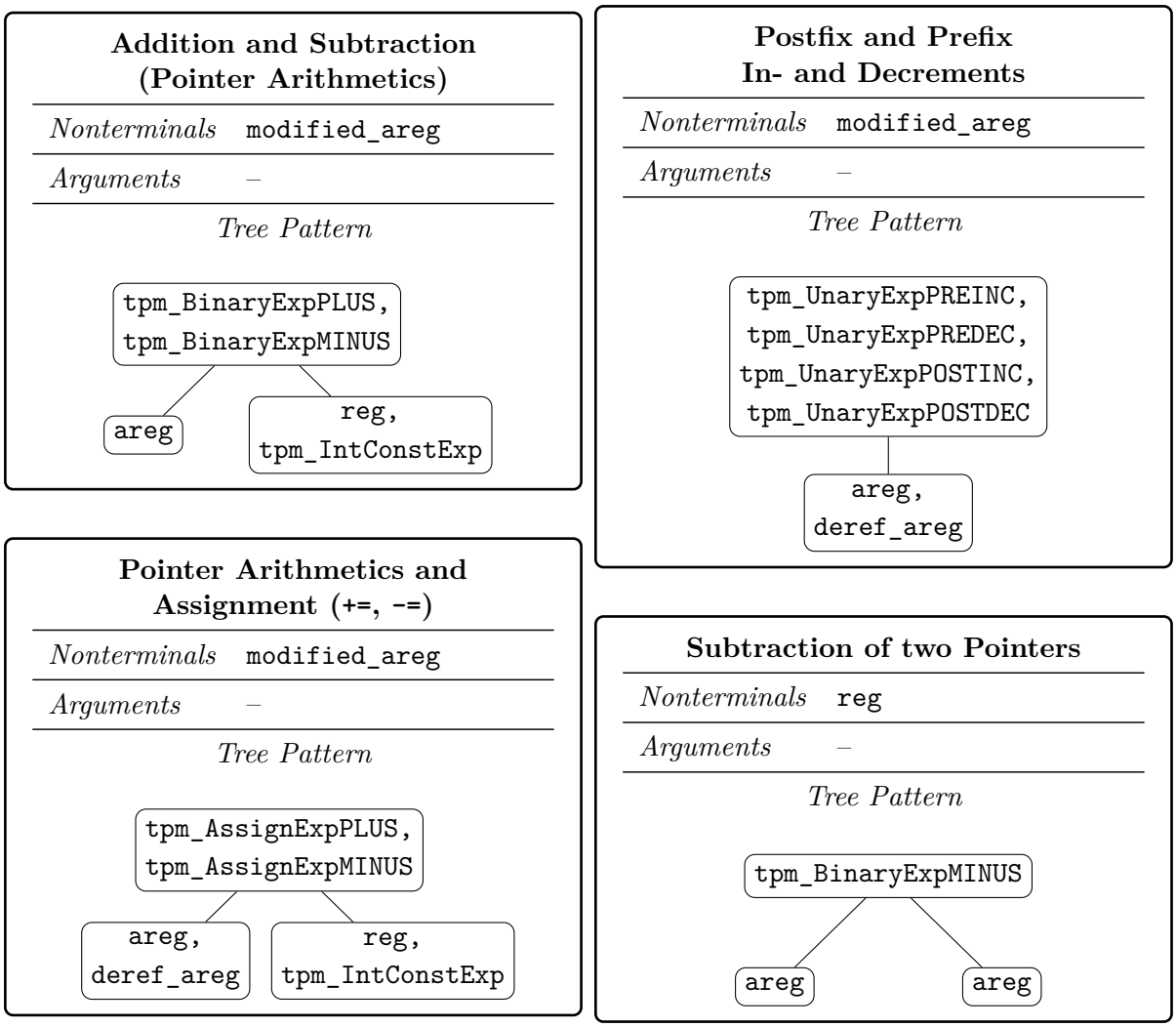


Figure 6.8.: Rules matching various expressions of *pointer arithmetics*

which was discussed at the end of section 6.3.2, only a single *store variant* [STRXX] would be inserted.

6.5.6. Pointer Arithmetics

Figure 6.8 presents various rules that can all broadly be categorized under the term *pointer arithmetics*. First, there is the *addition* (+) and *subtraction* (-) of pointers with *integer values*. Similarly, there are expressions combining *assignments* with *addition* (+=) or *subtraction* (-=). The various *prefix and postfix increments and decrements* (++p, p++, --p, p--) are also part of the collection. Finally, there is also the form of *subtraction* where two pointers are subtracted from one another yielding an *integer result*.

With the exception of *pointer subtraction*, every one of these rules returns the `modified_areg` **nonterminal**. Of all the rules returning the `modified_areg` **nonterminal**, not one inserts a single instruction. Instead, the **action**-parts package the modification represented by their

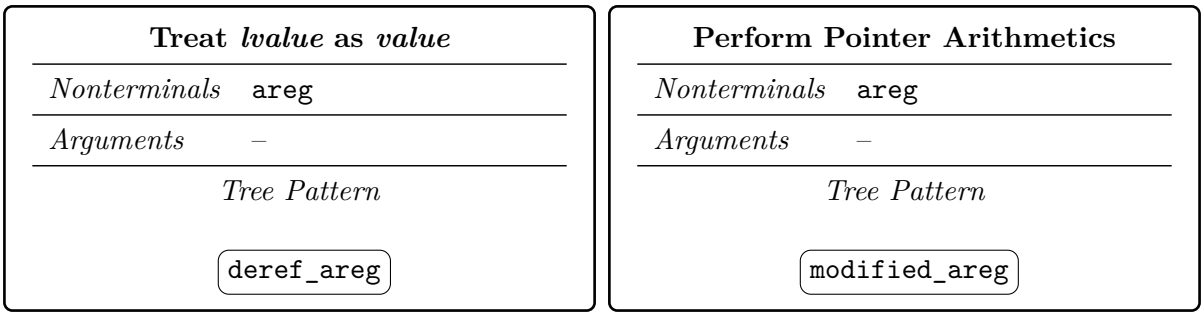


Figure 6.9.: Two special *chain rules*

expression into an object of type `AddressModification` which is then returned. Therefore, actual execution of pointer arithmetics is postponed to a later point. All of the rules returning the `modified_areg` **nonterminal** only contain the **cost** of their child-nodes.

The outlier among the set is *pointer subtraction*. Instead of taking a *pointer* and an *integer value*, this expressions takes two *pointers* of the same type and returns an *integer value* representing the number of *base type elements* between the two pointers. The pointers are represented by `areg` **nonterminals**, whereas the result is stored in a simple *value register* `reg`.

The **action**-part of this rule starts by evaluating both operands, beginning with the *left-hand side*. Then a *subtraction* [`SUB`] is inserted, subtracting both pointers and storing the result in a new *virtual register*. At this point the **action**-part checks whether the size of the base type is a power of two. If it is, the freshly computed result is *arithmetically shifted right* to realize the division by the size of the base type, meaning a negative result will preserve its sign, which is intended.

If the size of the base type is not a power of two, it is loaded into a register and division is performed at runtime. Unfortunately, the ARMv4 architecture does not feature a native instruction performing integer division. Therefore, an external library function has to be called that will perform the division, which is expensive due to the numerous instructions contained therein as well as the cost of the subroutine invocation itself, creating a new stack frame and pushing necessary registers on to the stack.

The **cost**-part of this rule always accounts for the cost of the child-node as well as the cost of the *subtraction* [`SUB`]. If the size of the base type is a power of two, the cost of the *move* [`MOV`] with the *shifter operand* is added. Otherwise, the cost of the *branch and link* [`BL`] instruction used to invoke the external subroutine is added This is an inaccurate representation of the actual cost involved in that route, but is used across the ruleset whenever an external library function is invoked.

6.5.7. Chain Rules

Figure 6.9 presents two special rules that both serve very important purposes. They are both *chain rules*, meaning they only take a single **nonterminal** as input and produce another **nonterminal** as output.

The *chain rule* presented on the left side of figure 6.9 transforms a `deref_areg` into an `areg`. An equivalent rule already exists for the `reg` and `deref_reg` **nonterminals**. There are many rules inside of the *code selector* that only take *values* as their inputs. Instead of writing every rule twice, once for the `deref_*` variant and once for the simple variant, this chain rule transforms the *lvalue* representation into that of a simple *value*. The rule's **cost**-part checks whether the expression it represents constitutes an assignment to the *lvalue*. If it does the rule returns a cost of *infinity*. Otherwise, the rule allows the transformation and merely contains the cost of its child-node.

The *chain rule* presented on the right side of figure 6.9 is an important part of the ruleset for pointer expressions. The reason for postponing the evaluation of pointer arithmetics was already discussed thoroughly over the past pages. There are, however, occasions where the pointer arithmetics cannot be further postponed or merged with a memory access. In those cases, the *pending modification* of the `modified_areg` **nonterminal** can be evaluated immediately.

This rule's **action**-part invokes the `AddressModification`'s `performModification` function which was discussed in subsection 6.3.1. The result is a `LLIR_Register*` containing the result of the modification, which is the `areg` **nonterminal** returned by this rule. Its **cost**-part includes the cost of its child-node as well as the worst-case estimate of instructions inserted when calling `performModification`.

6.5.8. Chained Address and Indirection Expression

Figure 6.10 presents the four rules related to the special *chained address and indirection expression* (`&*p`). The C standard mandates that no code shall be inserted for this kind of expression. To realize this, rules must be created that *bypass* the already existing rules matching the *address* and *indirection operator*.

The rules on the left side of figure 6.10 "box" the `areg` or `modified_areg` **nonterminal** into the `zero_op_areg` or `zero_op_modified_areg` **nonterminal** respectively. In doing so, the *indirection operator* (`*`) represented by the `tpm_UnaryExpDEREF` **terminal** is matched as well. These two rules insert no instructions and only report the cost of their child-nodes.

The two rules represented by the diagrams on the right side of figure 6.10 "unbox" the `zero_op_areg` or `zero_op_modified_areg` **nonterminal** returning the original `areg` or `modified_areg` **nonterminal**. In doing so, they match against the *address operator* (`&`)

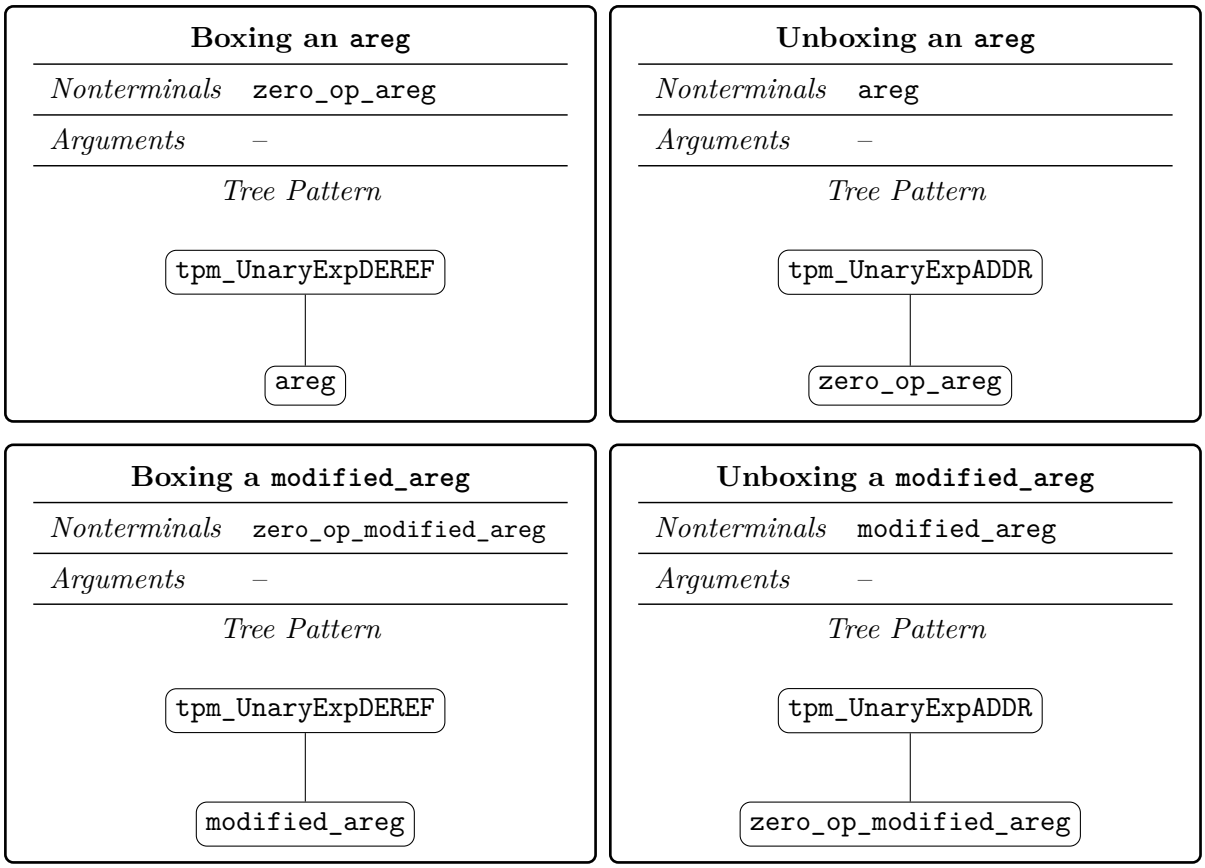


Figure 6.10.: Rules for *chained address and indirection operators* (`&*p`)

represented by the `tpm_UnaryExpADDR` **terminal**. And just as before these two rules also do not generate any instructions and simply pass through the cost of their child-node.

So far, the setup of these rules explains how they match against the *indirection* and *address operator* without generating any instructions. The key to these rules lies within the use of the specialized **nonterminals**. *Only* the "unboxing" rules match against the two new **nonterminals**, `zero_op_areg` and `zero_op_modified_areg`, and *only* the "boxing" rules return the two new nonterminals. In other words, matching against one mandates matching against its counterpart. But for that to work, the expression matched has to be the *chained indirection and address expression* (`&*p`).

Since all rules involved in the process incur no cost, the *tree pattern matcher* will always prefer these over the rules that actually generate instructions.

6.5.9. Miscellaneous Rules

Figure 6.11 presents the final set of rules that will be discussed in this section. Just like the rules shown in figure 6.9 both diagrams present *chain rules* as well.

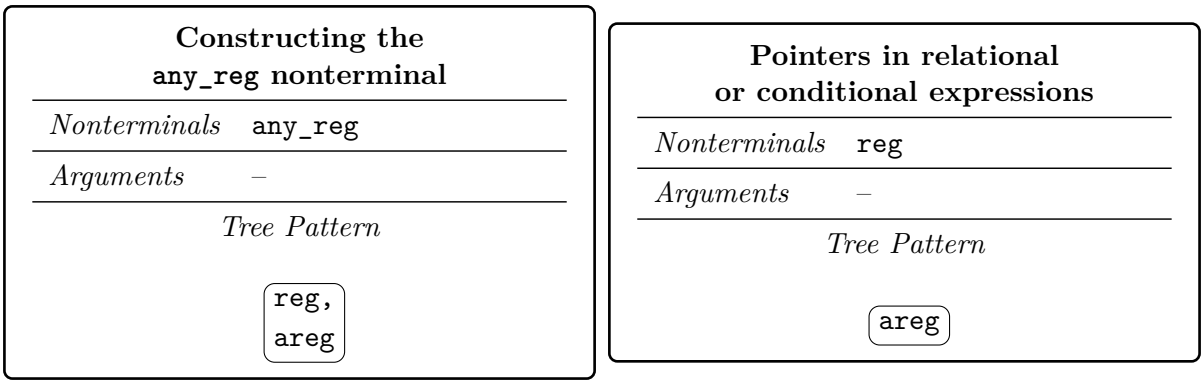


Figure 6.11.: Miscellaneous rules related to pointer expressions

The rule on the left side of figure 6.11 simply realizes the `any_reg` **nonterminal** by taking either the `areg` or the `reg` **nonterminal** as input, passing through their `LLIR_Register*` in the **action**-part and incurring no further **cost**.

The rule on the right side of figure 6.11 is unusual, since it breaks the clear divide between the `reg` and `areg` **nonterminals** by transforming the latter into the former. The **action**-part of this rule simply passes through the `LLIR_Register*` and generates no instructions.

The purpose of this rule is to accommodate *relational* and *conditional expressions* such as *less than* [`<`], *equality* [`==`] or *logical or* [`||`]. All of these expressions operate on *value registers*. However, when using *pointers* in these, the behavior is no different from that of simple *integer values*. Instead of reimplementing all of the already existing rules for *relational* and *conditional expressions*, this rule allows pointers to be used in the already existing ones.

This chain rule's **cost**-part manually checks whether the expression it represents is part of a *relational* or *conditional* expression. If that is indeed the case, the rule allows the recontextualization of the *address register* `areg` as a *value register* `reg`. Otherwise, it returns an *infinite* **cost**.

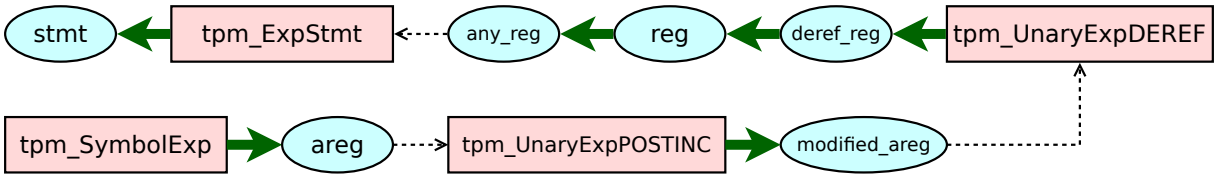
6.6. Example

Figure 6.12 shows a parsing example in which the **ICD-CG** translates a C statement to ARM assembly. Subfigure 6.12a shows the the input C statement on the left side of the subfigure and the generated instructions on the right side of the subfigure. Subfigure 6.12b, on the other hand, illustrates the **ICD-CG** *parse tree*. This tree includes the **terminals** given by the **ICD-C**, highlighted in red, and the **nonterminals** returned by the **ICD-CG**'s rules, highlighted in blue. Additionally, every application of an actual rule is highlighted by a large green arrow in the figure. Thin dotted arrows merely indicate a node being processed as the child node of a rule's **tree** pattern.

```
1 *(foo++);
```

```
1 LDR p1, [p0], #4
```

(a) The C statement (left) and the generated ARM instructions (right)



(b) The **ICD-CG** parse tree

Figure 6.12.: Example translation of a C statement with the developed ruleset

Incidentally, the parse tree presented in figure 6.12 is a special form of a tree, called a *path*, since none of the nodes have more than one child-node attached. A good way to untangle this chain of rule applications is by starting at the very bottom of the parse tree, with the **terminal** `tpm_SymbolExp`. It represents the symbol `foo` of the C statement, which is of type `int*`.

The first rule applied loads this pointer symbol. In the example provided foo is a *local register symbol*. This means the rule will return the symbol's corresponding `LLIR_Register*` packaged in the `areg` **nonterminal**, but no instructions will be inserted yet.

Next, one of the pointer arithmetic rules presented in subsection 6.5.6 matches against the `tpm_UnaryExpPOSTINC` **terminal** with an `areg` **nonterminal** as its child node. This rule does not generate any instructions either, but rather opts to postpone the pointer arithmetics to a later point by packaging the `LLIR_Register*` and the modification in an `AddressModification` instance. The modification described here is an increment (+1) with the *postfix write back policy*, since the *postfix increment operator* was used.

Now one of the indirection rules is applied, matching against the `tpm_UnaryExpDEREF` **terminal** with a `modified_areg` **nonterminal** as its child-node. As described in subsection 6.5.2, this rule packages the `AddressModification` instance into a `DerefInfo` instance. The `loadResult` parameter passed to this rule from the **action**-parts that supersede it is `true`. Therefore, `createLoad` is called on the `AddressModification` instance before it's packaged into the `DerefInfo`.

This method will then follow the behavior explained in subsection 6.3.2. In this case it will insert a *load word* [`LDR`] instruction since the base type of the pointer `foo` is `int`, which is 4 bytes wide. The result is loaded from `p0` which is the *virtual register* representing the symbol `foo`. *Post-indexing* is used to satisfy the *postfix write back policy* mandated in the `AddressModification` instance. The *final byte offset* inserted is 4 since the type size of an `int` is 4 and the specified *integer constant* was +1. The result is stored in `p1`, which is a *virtual register* freshly allocated by `createLoad` to store the result. The rule then embeds `p1` into the new `DerefInfo` instance as the `mResultRegister`.

The returned `deref_reg` **nonterminal** then undergoes two transformations. It is first recast as a simple `reg` since the `deref_reg` is not the destination of an assignment, i.e. its memory location is not of further relevance. It is then recast into the `any_reg` **nonterminal** so that it can be processed by rules that do not differentiate between *value* and *address registers*.

Finally, since the given expression constitutes the entirety of the statement, an already existing rule of the *code selector* matches against the `tpm_ExpStmt` **terminal** with the `any_reg` **nonterminal** as its child node. This is more of a formal step, no instructions are generated here. The rule returns the `stmt` nonterminal, which is the *final, single nonterminal* that concludes the *tree pattern matching* process.

7. Testing and Evaluation

This chapter tests the ruleset developed in chapters 5 and 6. There are three criteria that will be examined: **Correctness**, **code size** and **efficiency**. The first criterion will be the focus of section 7.1 where numerous testfiles have been written in a testbench framework to ensure the written rules are *correct*. Then, section 7.2 will evaluate four small benchmark files with respect to their *code size* and their *efficiency*, which is understood to refer to their *average* and *worst-case execution times*.

7.1. Testsuite

In order to evaluate the developed ruleset's **correctness**, the WCC's *testbench* is going to be used. Subsection 7.1.1 will introduce this *testbench*, explaining the API testfiles interface with as well as the actual steps performed during tests. Then, subsection 7.1.2 will print and explain one example testfile and showcase the output generated by the *testbench*. Finally, subsection 7.1.3 will summarize the large number of tests developed during this thesis, explaining what areas different groups cover.

7.1.1. The WCC's Testbench

The goal of the WCC's *testbench* is to check testfiles for their **correctness**. Properties such as *code size*, *efficiency* or even *energy consumption* are completely ignored here. Its general approach can be broken down into the following steps:

1. Compile the input testfile with the WCC and the `gcc`.
2. Simulate both *executable binaries* using the *GNU Debugger* (`gdb`).
3. Compare the results of both simulations. If they match, the test is considered a *pass*, otherwise it is considered a *fail*.
4. Repeat the above steps for all *optimization levels*. (`-O0` to `-O3`)

Of course, the compilation targets the correct architecture, in this case the ARM7 family of processors. For this thesis an additional environment variable must be set before invoking the testsuite:

Function	Arguments
check_i	signed int
check_c	char
check_pi	int*
check_ppi	int**
check_4i	int (x4)

Table 7.1.: Small excerpt of the functions made available by the *testbench* API

```
WCC_SUPP_CFLAGS="-mcodesel=wcc-codesel "
```

By default, the WCC is using the external `gcc` code selector as was explained in section 1.3. To make sure the testsuite uses the internal code selector where the new ruleset has been added, the above switch has to be provided.

The phrase ”compare the results“ used in the enumeration above raises two key questions: First of all, what exactly classifies as the *results* of our testfile? Moreover, how are these results, once identified, compared? This is where the *testbench*'s API comes into play.

Testbench API

Testfiles supplied to the *testbench* should be as self-contained as possible, using as few external libraries as possible, ideally none. Since the internal code selector is still a work in progress, many external libraries do not work at this point in time anyways. To use the *testbench*'s API the `checkio.h` header-file has to be included. This API provides a large collection of `check_*` functions, a small excerpt of which is presented in table 7.1.

These functions pass a value of a certain type to the simulator, which then prints a formatted version of the passed value to an internal buffer. There are `check_*` functions for every builtin type, including chained pointers. There even are functions provided for convenience which allow the programmer to pass multiple values of the same type with a single function call. These internal buffers where the passed values are stored are created during the simulation runs for both compiled binaries. It is precisely these buffers that are then compared to determine whether the test passed or not. Note that the return value of the program itself is also picked up by the simulator and printed to the buffer.

A testfile can invoke various `check_*` functions at crucial points in the program to verify that the `gcc`'s compiled binary produces the same result as the WCC's. Of course, this approach is not perfect. It relies on the assumption that the `gcc`, as a tried and tested compiler, will provide a *correct reference implementation*. But even the `gcc` has bugs that may result in incorrect results. Moreover, there are **a lot** of edge cases where the C standard leaves decisions in the hand of the

```
1  #include "checkio.h"
2
3  int main()
4  {
5    int a[] = {123, 53, 26, 73, 42, 35, 67};
6
7    int *p = a;
8    int *q = p + 5;
9
10   check_i( *p );
11   check_i( *q );
12
13   check_i( p - q );
14
15   return 0;
16 }
```

Listing 7.1: Example testfile for the *testbench* – `pointer_misc_01.c`

implementation. Two different compilers may then produce different results even if they both are standard-compliant because they chose to implement certain edge cases differently.

Therefore, this approach is neither complete nor does it prove *correctness* in the formal sense of the word. Rather, the *testbench* is an immensely useful tool to track down errors and bugs during development, and provide at least a certain sense of security that the ruleset behaves as expected.

7.1.2. Example Testfile

Listing 7.1 shows one of the many testfiles developed for the *testbench* during this thesis, `pointer_misc_01.c`. This *miscellaneous pointer test* first allocates an `int`-array called `a` with seven elements in line 5. Lines 7 and 8 then create two new pointers, `p` and `q`, to the first and sixth element of the array respectively.

The *testbench*'s API is used for the first time in lines 10 and 11. Here, the program asserts that the correct values are retrieved when dereferencing pointers `p` and `q`. Line 13 checks for the number of elements between the two pointers, using the *pointer subtraction* discussed in subsection 6.5.6. Finally, the program returns value 0 upon exit.

Listing 7.2 shows the contents of buffers created while simulating both compilations. Lines 3 and 8 present the first value that was put into the buffer for both simulation runs, in this case 123. This matches precisely with the first element of the `int`-array `a` that pointer `p` was supposed to point to. Similarly, lines 4 and 9 showcase the second item in the buffer, in this case the sixth element of the `int`-array, as expected.

```
1  Program exited normally.
2   **** Reference simulation output ****
3       ?0: 123
4       ?1: 35
5       ?2: -5
6       ?3: 0
7   **** Match simulation output ****
8       ?0: 123
9       ?1: 35
10      ?2: -5
11      ?3: 0
12  Success.
```

Listing 7.2: Simulation output for both compilations of `pointer_misc_01.c`

The third buffer line displays the number of elements between the two pointers determined through *pointer subtraction*. Since `p` was subtracted from `q` and not the other way around the result is negative, which behaves exactly as intended as well.

Finally, the last element of the buffer contains the program's return value, 0. Since the contents of both buffers match perfectly, this test was considered a success by the testsuite.

7.1.3. Developed Set of Tests

Table 7.2 provides an overview of the 126 *testbench* testfiles that were put together during this thesis to test the ruleset for **correctness**. Note that the WCC already supplies a large collection of tests for all of the different syntactic constructs that need to be considered. Many of the tests listed in the table have been adapted from this collection, with modifications applied to them since the internal code selector is currently incomplete.

The first section of the table concerns *jump statements*, specifically `break`, `continue` and `goto` statements, with a total of 13 testfiles provided for this group. The tests include different loop statements (`for`, `while`, `do-while`), in simple and nested forms. For the `goto` testfile various jumps were placed inside of a `for` loop.

The next section of table 7.2 tests section 5.2's rules for `switch` statements. Initially, six testfiles were adapted for the `signed int` type. These test for `switch` statements inside of `for` loops, sometimes using negative or *very large* values inside of the controlling expression and the `case` labels. These six tests were then cloned, with the type of the controlling expression exchanged for the different *builtin integer types* that have to be supported. Tests for these smaller types also check the behavior at the boundaries of the value ranges.

Finally, the largest set of *testbench* testfiles has been developed for the pointer expressions since these present the biggest challenge tackled in this thesis. First, there are twelve small

Testfile Name	Tests per Type	Description
	Jump statements	
`break_int_*.c`	6	Tests for the `break` statement.
`continue_int_*.c`	6	Tests for the `continue` statement.
`goto_int_*.c`	1	Test for the `goto` statement.
	`switch` *statements*	

`switch_char_*.c`
`switch_uchar_*.c`
`switch_short_*.c`
`switch_ushort_*.c`
`switch_int_*.c`
`switch_uint_*.c`
	6	Tests for the `switch` statement with various types used in the *controlling expression*.

	Pointer expressions	
`pointer_misc_*.c`	12	Miscellaneous pointer testfiles.
`pointer_void_*.c`	5	Testfiles for `void*`.

`pointer_char_*.c`
`pointer_uchar_*.c`
`pointer_short_*.c`
`pointer_ushort_*.c`
`pointer_int_*.c`
`pointer_uint_*.c`
	7	Pointer tests with various base types.

`pointer_char_array_*.c`
`pointer_uchar_array_*.c`
`pointer_short_array_*.c`
`pointer_ushort_array_*.c`
`pointer_int_array_*.c`
`pointer_uint_array_*.c`
	1	Testfiles combining pointers and arrays for various base types.

`pointer_char_struct_*.c`
`pointer_uchar_struct_*.c`
`pointer_short_struct_*.c`
`pointer_ushort_struct_*.c`
`pointer_int_struct_*.c`
`pointer_uint_struct_*.c`
	2	Testfiles combining pointers and `struct` types for various base types.

Table 7.2.: Categories of *testbench* testfiles developed for this thesis

miscellaneous tests that were put together while chapter 6's ruleset was developed to debug the written code. These are small and limited in scope and are included here for completeness' sake.

Similarly to the tests for `switch` statements, seven tests have been developed for `signed int` and (chained) pointers to `signed int`. These tests check, among other things, for correct behavior during dereferencing, make sure pointers can be correctly passed as function arguments and can be used correctly with relational operators such as *equality* [`==`] or *less than* [`<`]. Of course there are also tests dedicated to checking various forms of *pointer arithmetics* as well as the use of *chained pointers*, i.e. pointers pointing at pointers, such as `int**`.

There is also a test making sure arrays and pointers harmonize like one would expect, with array symbols "decaying" to pointers. Two tests combine `struct` types with pointers and test for various operations when using pointers to and within those `struct` types.

These tests have then been cloned for the other *builtin integer types* as well to ensure they behave correctly. This is especially important since a different *base type* prompts the use of a different *load and store variant* which in turn influences the applicable *addressing mode*.

Finally, five tests have been made specially to test for `void*`. This is a special type of pointer that cannot be dereferenced but behaves like a `char*` during pointer arithmetics. It has to be cast to a different pointer type before the *indirection operator* can be applied.

All of the developed tests successfully pass the testsuite on all optimization levels [1]. The complete collection of *testbench* testfiles can be found on the accompanying CD. Please refer to chapter A in the appendix for more information.

7.2. Evaluation

This section will evaluate the written ruleset with respect to its **code size** and **efficiency**, which is measured using a program's *average* and *worst-case execution time*. First, subsection 7.2.1 will outline the general approach used for each small benchmark, including the compilations and analyses performed on the benchmark and what aspects will receive special focus. Then, each of the four remaining sections will be dedicated to one of those small benchmarks, with subsection 7.2.2 examining *jump statements*, subsection 7.2.3 investigating `switch` statements with `break` and `goto` statements, while subsections 7.2.4 and 7.2.5 will examine pointers in conjunction with arrays and `struct` types respectively.

[1] Note that, because of recent changes made to the *register allocator*, some tests may currently fail. This is not due to errors in the ruleset, but rather due to bugs in the register allocator that were introduced with aforementioned changes.

7.2.1. Methodology

In each of the following subsections, a small C benchmark file will be examined. The testfile as a whole will be explained briefly, but the bulk of the attention is put towards the expressions that were implemented as part of the ruleset in chapters 5 and 6.

These small benchmarks will be compiled twice: Once with the WCC utilizing the `gcc`'s code selector, and once with the WCC utilizing the *internal* code selector where the new rules have been added. Ideally, both compilations, including the `gcc`'s invocation when compiling with its code selector, would be done on *optimization level* `-O0`, deactivating all optimizations so that only the code selector's performance is evaluated. Unfortunately, as explained below, this would lead to inaccurate results. While the WCC itself will be invoked with *optimization level* `-O0`, the invocation of the `gcc` when using its code selector is performed with the `-O1` switch.

Recently, a *register allocator* utilizing a *graph-coloring heuristic* has been implemented for the WCC's ARMv4 workflow, replacing the previously used *spill-all register allocator* that simply spilled all virtual registers on the stack. This improves the performance of generated binaries significantly. When invoking the `gcc` with *optimization level* `-O0`, an inefficient register allocation algorithm is used that does not compare with the results achieved by the WCC's *graph-coloring-based register allocator*. To level the playing field, the `gcc` is invoked with optimization level `-O1`, which causes a more efficient *register allocation algorithm* that is comparable to the WCC's *graph-coloring-based* solution to take over.

While this approach solves the *register allocation* problem, it introduces a new one: The `gcc` will now apply various optimizations included in the `-O1` *optimization level* that are not performed by the WCC since it is invoked with `-O0`. The goal of this evaluation is not to compare the two compilers' optimizations. Therefore, various `-fno-*` switches are passed to the `gcc`, manually deactivating many of the optimizations included in *optimization level* `-O1`. Unfortunately, as will become clear when evaluating the small benchmarks later, the `gcc`'s code selector will still apply some transformations that could not be deactivated through an invocation-flag.

The resulting ARM assembly for both compilations is presented in each subsection. Note that these assembly excerpts have been edited. Meta-information that does not represent actual instructions has mostly been cut to keep the listings focused, and many labels have been renamed to increase readability. The accompanying CD includes the C sources of the small benchmarks as well as the full, unedited compilations created by the WCC and the `gcc`. Please refer to chapter A of the appendix for more information.

The assembly listings are then examined, highlighting which assembly instructions realize the C expressions that are the focus of the benchmark. The different approaches taken by the `gcc` compared to the developed ruleset are analyzed. Note that not every assembly instruction of the listing will be explained in depth. Rather the focus will be put on those instructions corresponding to the expressions of the ruleset.

```
1  int main() {
2    int x = 0;
3
4    for ( int i = 0; i < 20; ++i ) {
5      if ( i < 4 )
6        continue;
7
8      x += i;
9    }
10
11   return x;
12 }
```

Listing 7.3: `jumps.c` – Simple `for` loop utilizing a `continue` statement

Finally, three properties of the two compilations will be compared:

- The **Average Case Execution Time (ACET)** will be determined through simulation by *CoMET*.
- The **Worst Case Execution Time (WCET)** will be estimated by `aiT` through *static timing analysis*.
- The **Code Size** of the two compilations can be read off of the generated assembly and is given in *bytes*. It directly correlates to the number of instructions present.

The results of these analyses will be presented in a small table in every subsection. Moreover, the notable discrepancies (or lack thereof) between the different metrics will be explained with the aid of the assembly listings.

7.2.2. `for` loop with `continue` Statement

The first small benchmark examined is given in listing 7.3. It's sole purpose is to test the implementation of the `continue` statement. The local integer variable `x` initialized in line 2 is incremented by the current loop index every time the `for` loop's body ranging from line 4 to 9 is executed. However, if the current loop index is smaller than 4, line 6 will cut the current loop iteration short, skipping `x`'s addition in line 8.

Listing 7.4 showcases the `gcc`'s compilation. It transformed the `for` loop of the C source into a `do-while` loop, evaluating the *conditional expression* at the bottom of the loop, not at the top. This technique is called *loop inversion* [32] and marks one of the optimizations performed by the `gcc` on *optimization level* `-O1` that could not be deactivated. This transformation can be applied here because the loop is always entered, enabling a translation that uses only **one** conditional branch in line 12 of the assembly listing.

```
1  main:
2      mov ip, sp
3      stmfd sp!, {fp, ip, lr, pc}
4      sub fp, ip, #4
5      mov r0, #0
6      mov r3, #1
7  L2:
8      cmp r3, #3
9      addgt r0, r0, r3
10     add r3, r3, #1
11     cmp r3, #19
12     ble L2
13 LBE2:
14     ldmfd sp, {fp, sp, pc}
15     .size main,44
```

Listing 7.4: `gcc` translation of `jumps.c`

```
1  main:
2      mov ip, sp
3      stmfd sp!, {fp, ip, lr, pc}
4      sub fp, ip, #4
5      mov r0, #0
6      mov r2, #0
7  _L1:
8      @Loop condition: FOR
9      cmp r2, #20
10     bge _L2
11 _L3:
12     cmp r2, #4
13     bge _L4
14 _L5:
15     b   _L6
16 _L4:
17     add r0, r0, r2
18 _L6:
19     add r2, r2, #1
20     b   _L1
21 _L2:
22     sub sp, fp, #12
23     ldmfd sp, {fp, sp, pc}
24     .size main,60
```

Listing 7.5: WCC translation of `jumps.c`

Line 9 performs the loop body's central statement, `x += i;`. This *addition* [`ADD`] has a *conditional code* attached to it, only executing if the most recent *comparison* [`CMP`] resulted in the operand on the *left-hand side* being *greater than* [`GT`] the operand on the *right-hand side*. The *comparison* [`CMP`] in question precedes the *conditional addition*, located in line 8. It compares the contents of `r3`, the loop index, with immediate constant `#3`, realizing the `if` statement in line 5 of `jumps.c`. By utilizing *conditional execution* to select whether the addition `x += i;` should be executed, no branches have to be inserted for the `if` or the `continue` statement. The only *branch* present is the one in line 12 that realizes the transformed `for` loop's functionality.

The WCC's translation can be seen in listing 7.5 which uses a total of *four* branches, two conditional and two unconditional. Instead of using *conditional execution* to select whether the addition `x += i;` should be performed, branches are used instead. Lines 12 and 13 realize the `jumps.c`'s `if` statement from line 5 (`if ( i < 4 )`). If the *controlling expression* is *false*, the *conditional branch* [`BGE`] is taken, jumping directly to the loop body's addition. If it evaluates to *true* however, the conditional branch is skipped and the *unconditional branch* [`B`] of line 15 is taken instead. It is precisely this line that was inserted by the code selector for the `continue` statement, inserting a single *unconditional branch* [`B`] as was outlined in section 5.1. This unconditional branch jumps to label `_L6` at the end of the loop body. At this location the *loop index* `r2` is incremented and control-flow branches to the beginning of the loop at label `_L1`.

Code Selector	**ACET**	**WCET**	**Code Size (Bytes)**
WCC	1479	1490	72
`gcc`	843	854	56
Delta in %	75%	74%	29%

Table 7.3.: Evaluation results for `jumps.c`

Note that, unlike the `gcc`, the WCC's compilation did not transform the `for` loop into a `do-while` loop because optimizations were turned off. This leads to **two** branches inserted for the `for` loop, one in line 13 branching out of the loop if necessary, and one in line 20 jumping back up to the head of the loop.

Table 7.3 displays the results of the different evaluations. First of all, note that the ACET and WCET for both compilations are very close to each other. This is due to the lack of input and complexity in the test program. The static analyzer is therefore able to give a very tight estimate of the program's actual runtime.

The WCC's compilation is 16 bytes larger, due it having four additional instructions: One *subtraction* [`SUB`] in line 22 when returning and the three additional *branches* discussed earlier. Since these branches are part of the `for` loop's body they are executed multiple times, contributing greatly to the program's runtime. This effect can be seen in the ACET as well as the WCET, increasing the WCC compilation's runtime by roughly 75% compared to the `gcc`'s. Due to the *loop inversion* optimization performed by the `gcc`, the comparison is not entirely fair. Nonetheless, realizing the `if` and `continue` with the aid of *conditional execution* will produce a more efficient program than the branch-based approach taken by the WCC.

7.2.3. `switch` Statement with `break` and `goto` Jumps

Listing 7.6 shows the benchmark that was written to test the implementation for `switch` statements developed in section 5.2 as well as the use of `goto` and `break` statements therein. First of all, note that the `switch` statement that is tested in this benchmark is wrapped inside of a `for` loop which begins in line 4. This allows testing many different values for the `switch` statement's *controlling expression*. Similar to listing 7.3 there is a local integer variable `x` that is incremented over the course of the program.

The `switch` statement itself uses the loop index `i` as its controlling expression and lists five destination labels:

- `case 3`: Increase `x` by 56 and fall through to the next `case` block, `case 5`.
- `case 5`: Increase `x` by 12 and `break` outside of the `switch` statement.
- `case 27`: Increase `x` by 1 and `break` outside of the `switch` statement.
- `case 42`: Increase `x` by 123 and jump to label `foolabel`, which marks the same destination as `case` label `case 3`.

```
1  int main() {
2    int x = 0;
3
4    for ( int i = 0; i < 100; ++i ) {
5      switch ( i ) {
6        case 3:
7        foolabel:
8          x += 56;
9        case 5:
10         x += 12;
11         break;
12       case 27:
13         ++x;
14         break;
15       case 42:
16         x += 123;
17         goto foolabel;
18       default:
19         x += 5;
20     }
21   }
22
23   return x;
24 }
```

Listing 7.6: `switch.c` – `switch` statement utilizing `break` and `goto` jumps

- `default`: The `default` case increases `x` by 5. Since the `default` label was specified last, no `break` is necessary to break out of the `switch` statement after increasing `x`.

Listing 7.8 shows the WCC's translation of the `switch.c` benchmark. The control flow implied by the C source matches the control flow that can be observed in the translated ARM assembly very closely. Realization of the `switch` statement begins in line 12. Here, the chain of *test equality* [`TEQ`] and *conditional branch* [`BEQ`] instructions can be seen, where `r2`, the loop index, is compared against the different `case` constants. If a match has been found control flow will branch to the respective destination label.

Line 24 then performs the *unconditional branch* [`B`] to the default label, which corresponds to label `_L12` in the assembly source. The two `break` statements of the C source can be found in lines 29 and 32 in the ARM assembly, realized by *unconditional branches* [`B`] to label `_L14`. Finally, the `goto` statement's *unconditional branch* [`B`] can be found in line 35. Figure 7.1 visualizes the WCC's realization of the `switch` statement in a flowchart. The previously mentioned "chain" of equality checks is clearly visualized here. Only if all of the checks fail will the `default` branch be taken.

The `gcc`'s translation of the small benchmark is shown in listing 7.7. Note that, generally the `gcc` will attempt to translate `switch` statements with so-called *branch tables*. When using this

```
main:
    mov ip, sp
    stmfd sp!, {fp, ip, lr, pc}
    sub fp, ip, #4
    mov r0, #0
    mov r3, r0
L2:
    cmp r3, #5
    beq L5
L2_S1:
    bgt L8
L2_S2:
    cmp r3, #3
    beq L4
L2_S3:
    b   L3
L8:
    cmp r3, #27
    beq L6
L8_S4:
    cmp r3, #42
    beq L7
L8_S5:
    b   L3
L4:
    add r0, r0, #56
L5:
    add r0, r0, #12
    b   L9
L6:
    add r0, r0, #1
    mov r3, #28
    b   L2
L7:
    add r0, r0, #123
    b   L4
L3:
    add r0, r0, #5
L9:
    add r3, r3, #1
    cmp r3, #99
    ble L2
LBE2:
    ldmfd sp, {fp, sp, pc}
    .size main,116
```

Listing 7.7: `gcc` translation of `switch.c`

```
main:
    mov ip, sp
    stmfd sp!, {fp, ip, lr, pc}
    sub fp, ip, #4
    mov r0, #0
    mov r2, #0
_L1:
    @Loop condition: FOR
    cmp r2, #100
    bge _L2
_L3:
    teq r2, #3
    beq _L13
_L5:
    teq r2, #5
    beq _L6
_L7:
    teq r2, #27
    beq _L8
_L9:
    teq r2, #42
    beq _L10
_L11:
    b   _L12
_L13:
    add r0, r0, #56
_L6:
    add r0, r0, #12
    b   _L14
_L8:
    add r0, r0, #1
    b   _L14
_L10:
    add r0, r0, #123
    b   _L13
_L12:
    add r0, r0, #5
_L14:
    add r2, r2, #1
    b   _L1
_L2:
    sub sp, fp, #12
    ldmfd sp, {fp, sp, pc}
    .size main,112
```

Listing 7.8: WCC translation of `switch.c`

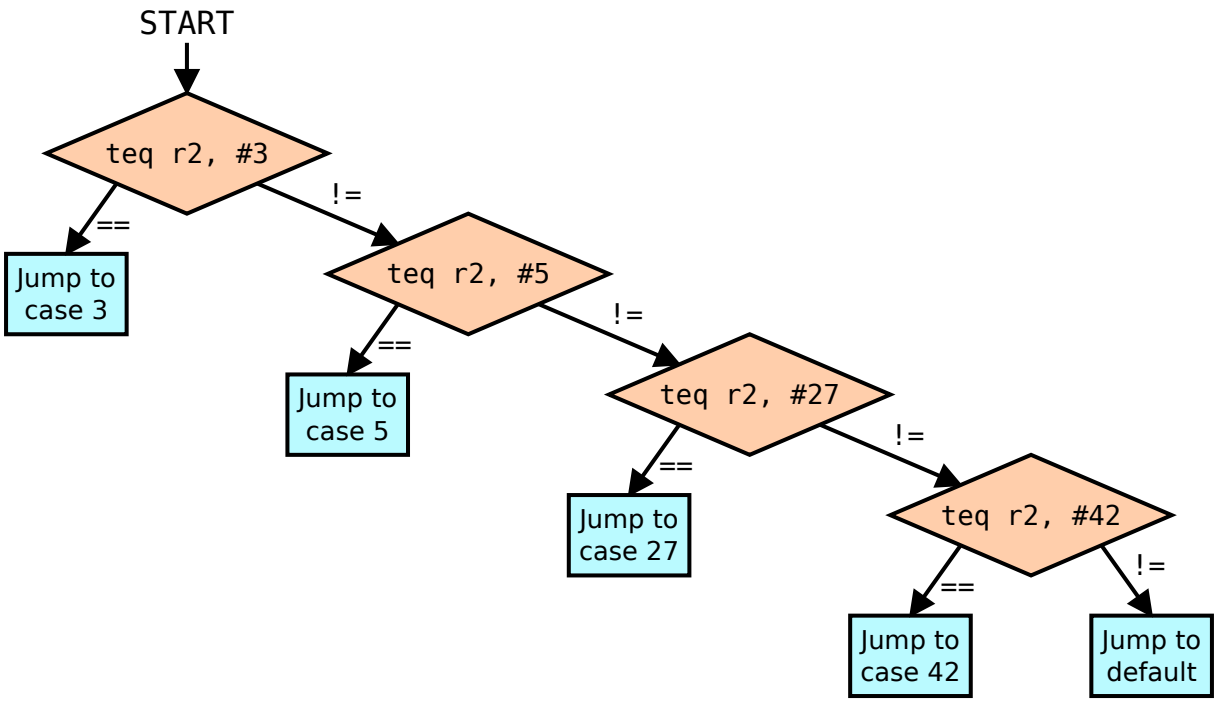


Figure 7.1.: Flowchart for the WCC's realization of the `switch` statement

strategy, the destination addresses are stored in a *branch table* in memory. The *controlling expression* is then used as an offset when accessing the *branch table*, causing the correct destination label to be read from the table. This works especially well if all the `case` labels are numerically close to each other, easing the generation of the *branch table*. [33]

The crucial advantage of this approach is the constant evaluation time. Reading the destination address from memory and jumping there takes the same time no matter the value contained in the *controlling expression*. This is decidedly different from the solution developed in section 5.2, where the evaluation time depends on the *controlling expression*'s value. There are three reasons why the WCC's implementation does not use *branch tables*:

1. The **ICD-LLIR** framework does not allow for a *low-level basic block* to have more than two successors. This is a technical limitation of the framework used, and would prevent us from inserting a *branch* instruction with a dynamic destination address, since this instruction's basic block could have far more than just two successors.
2. *Branch tables* make *static timing analysis* more difficult because the analyzer has to infer the set of possible destinations from a section in memory. If the `case` labels are numerically further apart, the *branch table* may contain a lot of "gaps" that are filled with the *default destination address*. This increases the size of the *branch table* and therefore the amount of potential addresses that a static timing analyzer has to consider.
3. Finally, *branch tables* can present a security hazard. Since the destination of *control flow* is read from memory, a malicious entity may attempt to overwrite the addresses stored there to take over the program, similar to a *buffer-overflow* attack that attempts to overwrite a

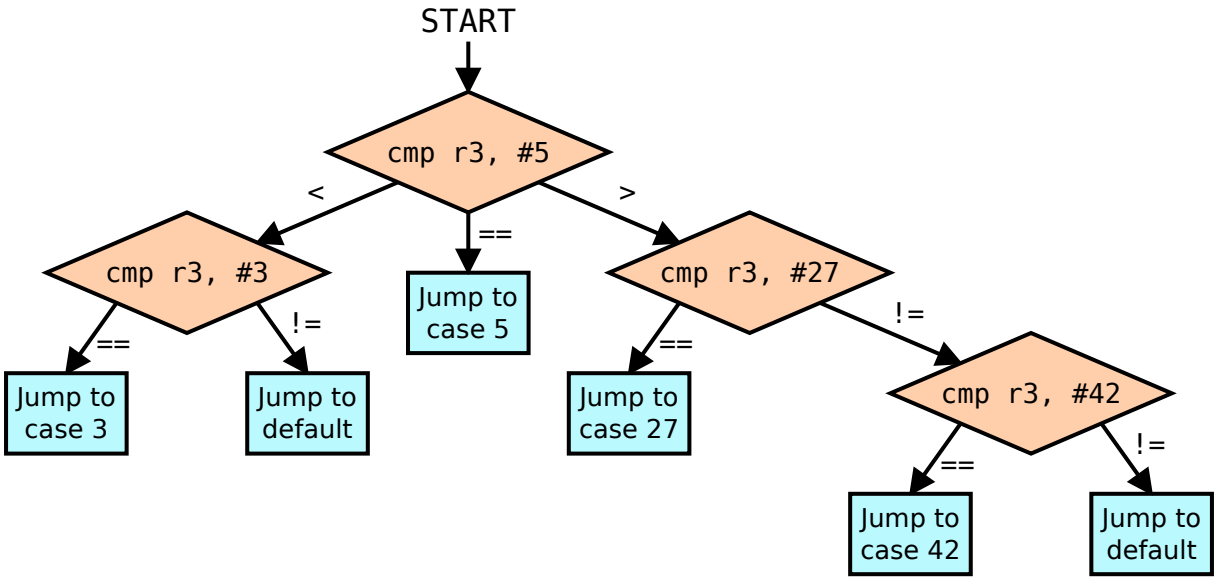


Figure 7.2.: Flowchart for the `gcc`'s realization of the `switch` statement

stack-frame's return address.

To make the evaluation fair, generation of *branch tables* has been explicitly forbidden when invoking the `gcc` code selector. This causes the `gcc` to realize the `switch` statement with a collection of *comparisons* and *branches*, similar to the WCC's translation. Even then, there are notable differences in the `gcc`'s approach that will be discussed next.

Since there are a lot of labels, comparisons and branches, the `gcc`'s realization of the switch statement will not be discussed directly with the assembly source, but rather with the help of the flowchart shown in figure 7.2. The comparisons and branches presented in this flowchart match the instructions given in the listing.

When comparing the two approaches for realizing the `switch` statement shown in the two figures, one important difference becomes clear immediately: While the WCC uses a *chain* of comparisons to determine the correct destination label, the `gcc`'s collection of comparisons and branches form a *tree*. The very first *comparison* [`CMP`] can have three different results, depending on whether the *controlling expression* was smaller than, larger than or equal to the immediate constant `#5`. This different approach results in one additional *branch* [`B`] to the default label as well as one additional *branch* [`B`] not shown in the figure that helps realize the *tree* structure.

Generally, the *tree* will perform better than the *chain* with respect to runtime. Imagine a `switch` statement with at least one hundred `case` labels. If a *binary tree* of comparisons is used there are at most $log_2(100) \approx 7$ *layers* of comparisons that would have to be executed by the target machine. However, if a simple chain of comparisons is used instead, there are 100 layers of comparisons that potentially have to be executed before reaching the correct destination label. The problem that manifests itself here is very similar to the problem of locating a specific element in a sorted list.

Code Selector	**ACET**	**WCET**	**Code Size (Bytes)**
WCC	10899	10910	112
`gcc`	10731	10742	116
Delta in %	2%	2%	-3%

Table 7.4.: Evaluation results for `switch.c`

```
1  int main() {
2    int a[100];
3
4    int *q = &*a + 57;
5
6    for ( int i = 0; i < 100; ++i )
7      *(a + i) = q - (a + i);
8
9    return 0;
10 }
```

Listing 7.9: `arrayPointer.c` – Evaluation file combining pointers and arrays

The small benchmark examined here is very short, however, not using hundreds of `case` labels but rather only five. Therefore, the theoretical difference in timing between these two approaches does not yet occur practically. All measured metrics shown in table 7.4 are very close to each other, with the WCC's timing only being marginally worse than that of the `gcc`.

7.2.4. Pointers and Arrays

Listing 7.9 is the first of two benchmarks testing *pointer expressions*. Specifically, this listing tests for the combined use of arrays and pointers. Central to this testfile is the `int`-array `a` containing 100 integers. In line 4 an `int*` named `q` is initialized pointing to the 58th element of the array `a`. In this expression pointer arithmetics as well as the *chained address and indirection operator* discussed in section 6.5.8 is used.

The heart of this small benchmark lies within line 7 which is embedded in a `for`-loop running its loop counter `i` from 0 to 99. Here, a new value is assigned to the `(i+1)`-th element of the `int`-array `a`. This value is the result of *pointer subtraction*. Specifically, the address of the `(i+1)`-th array element is subtracted from the previously initialized `int* q`. For the very first element this subtraction would, for instance, yield the value +57, for the 60th element it would yield −2 and for the 100th element it would yield −42.

Listing 7.11 shows the WCC's translation of benchmark `arrayPointer.c`. The initialization of `int* q` takes place in line 6 of the listing. Since the `int`-array `a` resides at `sp+0`, the *stack pointer* [`sp`] can used directly in the addition (`&*a = 57`). Note that, as discussed in subsection

```
1  main:
2      mov ip, sp
3      stmfd sp!, {fp, ip, lr, pc}
4      sub fp, ip, #4
5      sub sp, sp, #400
6      mov ip, #0
7  L2:
8      mov r0, ip,  asl #2
9      sub r1, fp, #412
10     add r2, r1, r0
11     sub r3, fp, #184
12     rsb r3, r2, r3
13     mov r3, r3,  asr #2
14     str r3, [r1, r0]
15     add ip, ip, #1
16     cmp ip, #99
17     ble L2
18 LBE2:
19     mov r0, #0
20     sub sp, fp, #12
21     ldmfd sp, {fp, sp, pc}
22     .size main,72
```

Listing 7.10: `gcc` translation of `arrayPointer.c`

```
1  main:
2      mov ip, sp
3      stmfd sp!, {fp, ip, lr, pc}
4      sub fp, ip, #4
5      sub sp, sp, #400
6      add r2, sp, #228
7      mov r0, #0
8  _L1:
9      @Loop condition: FOR
10     cmp r0, #100
11     bge _L2
12 _L3:
13     add r3, sp, r0,  lsl #2
14     sub r3, r2, r3
15     mov r3, r3,  asr #2
16     str r3, [sp, r0,  lsl #2]
17     add r0, r0, #1
18     b   _L1
19 _L2:
20     mov r0, #0
21     sub sp, fp, #12
22     ldmfd sp, {fp, sp, pc}
23     .size main,68
```

Listing 7.11: WCC translation of `arrayPointer.c`

6.5.8, no instructions are generated for the `&*a` expression. The inserted *addition* [`ADD`] adds the correct *byte offset* of $57 \cdot 4 = 228$ to the *stack pointer* [`sp`] and stores the result in `r2`.

The *pointer subtraction* of the C source (`q - (a + i)`) is realized by the WCC in lines 13 to 15. In line 13 the expression `a + i` is evaluated. The `int`-array `a` is located at the *stack pointer* [`sp`] while the loop counter `i` is stored in `r0`. Since the pointer has to be modified by increments of the size of its *base type*, a *logical shift left* is performed on the loop counter, effectively multiplying the value by four. Line 14 then performs the actual *pointer subtraction*, which stores the *byte difference* in register `r3`. Since pointer subtraction has to return the *number of base type elements between the two types*, this result is then *arithmetically shifted right* in line 15.

Finally, line 16 of the WCC's translation realizes the assignment of the result (`*(a + i) = ...`). The `int`-array `a` residing at the *stack pointer* [`sp`] is offset by the loop counter `i`. Since the integer offset given by `i` has to be applied as a *byte offset* it is *logically shifted left* by two bits. The result of the previous pointer subtraction residing in register `r3` is the *source value* that is written to memory by this instruction.

Listing 7.10 prints the `gcc`'s translation of the benchmark. Since the value of `int* q` is used only once in line 7, the `gcc` decided to shift the initialization (and therefore the definition) of `q` to line 11. Note that the `gcc` uses *frame pointer* [`fp`]-relative addressing instead of the *stack*

Code Selector	**ACET**	**WCET**	**Code Size (Bytes)**
WCC	6199	6210	68
`gcc`	7357	7368	72
Delta in %	-16%	-16%	-6%

Table 7.5.: Evaluation results for `arrayPointer.c`

pointer [`sp`]-relative addressing used by the WCC. Since the *frame pointer* [`fp`] resides at the top of the *stack frame* the calculated *byte offset* is different from that used by the WCC, on top of the fact that *subtraction* [`SUB`] instead of *addition* [`ADD`] takes place. The `gcc`, too, does not insert any instructions for the `&*a` expression.

The expression `a + i` of `arrayPointer.c`'s line 7 is evaluated by lines 8 to 10 of the ARM assembly. Line 8 transforms the loop counter `ip` to the *byte offset*, *arithmetically shifting left* the register by two bits, and stores the result in `r0`. Since the `gcc` uses *frame pointer* [`fp`]-relative addressing, `int`-array `a` resides at `fp - 412`. In line 9 this address is evaluated with a *subtraction* [`SUB`] and stored in register `r1`. Line 10 actually offsets the address pointing to `a`, register `r1`, by the correct *byte offset* residing in `r0`, storing the result in `r2`.

The `gcc` then realizes the *pointer subtraction* in lines 12 and 13. The previously calculated address of `a + i` available in `r2` is subtracted from the value of `q` residing in `r3`. Since the *number of base type elements* between the two pointers has to be returned, the `gcc` uses an *arithmetic shift right* by two bits in line 13 as well to divide the result by four. Finally, the *store* [`STR`] reuses the address of `a` residing in `r1` as well as the correct byte offset still present in `r0` to store the result in the correct memory location.

Note that the `gcc` has again applied the *loop inversion* optimization to the `for` loop, moving the *conditional expression* to the bottom of the loop, thereby inserting one *branch* [`B`] less than the WCC's translation.

Table 7.5 presents the evaluation results of benchmark `arrayPointer.c`. Curiously, the WCC's compilation performs better than that of the `gcc` in all categories, with a runtime that is roughly 16% faster than that of the `gcc`. In part, this is thanks to the bonus in efficiency provided by the *shifter operands* of *addressing modes 1* and *2*. In another part, however, this result is also down to some luck.

The `int`-array symbol `a` resides at `sp + 0` in the WCC's translation, whereas it resides at `fp - 412` in the `gcc`'s. Since addition with zero does not have to be performed, the WCC's implementation can save itself instructions that would add the appropriate byte offset to the *stack pointer* [`sp`] to access the array. Such an instruction is generated by the `gcc` and can be found in line 9, contributing to the measured time since this instruction is performed every time the loop body is executed.

```
1  struct foo {
2    char s[123];
3    int x;
4  };
5
6  int main() {
7    struct foo fa[100];
8
9    struct foo *p = fa;
10   struct foo *q = fa + 100;
11
12   while ( p < q )
13     (p++)->x = 42;
14
15   return 0;
16 }
```

Listing 7.12: `structPointer.c` – Evaluation file combining pointers and `struct` types

Another inefficiency introduced by the `gcc` is its moving of the initialization of `int* q`. Since the instruction calculating the address of `q` resides in line 11, it, too, is executed 100 times as part of the loop body. This is unnecessary, since the variable can easily be initialized before the loop, as is done by the WCC's translation.

However, when it comes to the actual "number crunching", the evaluation of pointer arithmetics like the *pointer subtraction*, both implementations are on par. The `gcc` and the WCC utilize *shifter operands* to realize the multiplication or division with the base type.

7.2.5. Pointers and `struct` Types

Listing 7.12 presents the last of the small benchmarks discussed during this evaluation, testing pointers referencing `struct` types. First of all, note the definition of `struct foo` in lines 1 through 4. It contains 123 characters packaged in the array `s` as well as a single integer `x`. Theoretically that would place its size at 127 bytes. However, both the `gcc` and the WCC align the `struct` members to *word*-boundaries, that is to *4 byte*-boundaries. Therefore, the size of `struct foo` when implemented grows to 128 bytes.

Central to this small benchmark is the large array `fa` of type `struct foo` containing 100 members. It is the reason for the very large size of the stack frame which can be seen in line 5 of *both* translations shown in listings 7.13 and 7.14, where the *stack pointer* [`sp`] is offset by 12,800 bytes. In lines 9 and 10 two variables of type `struct foo*` are initialized. `p` points to the first element of array `fa` while `q` points *one past the last element* of `fa`.

A `while`-loop in line 12 then repeats line 13 until pointer `p` is pointing to the same location pointed to by `q`. Iterating through all of the members of `fa` could also have been realized with a

```
1  main:
2      mov ip, sp
3      stmfd sp!, {fp, ip, lr, pc}
4      sub fp, ip, #4
5      sub sp, sp, #12800
6      sub r2, fp, #12800
7      sub r2, r2, #12
8  L2:
9      mov r3, #42
10     str r3, [r2, #124]
11     add r2, r2, #128
12     sub r3, fp, #12
13     cmp r2, r3
14     bcc L2
15 L2_S1:
16     mov r0, #0
17     sub sp, fp, #12
18     ldmfd sp, {fp, sp, pc}
19     .size main,60
```

Listing 7.13: `gcc` translation of `structPointer.c`

```
1  main:
2      mov ip, sp
3      stmfd sp!, {fp, ip, lr, pc}
4      sub fp, ip, #4
5      sub sp, sp, #12800
6      add r0, sp, #0
7      add r2, sp, #12800
8  _L1:
9      @Loop condition: WHILE
10     cmp r0, r2
11     bge _L2
12 _L3:
13     mov r1, #42
14     mov r3, r0
15     add r0, r0, #128
16     str r1, [r3, #124]
17     b   _L1
18 _L2:
19     mov r0, #0
20     sub sp, fp, #12
21     ldmfd sp, {fp, sp, pc}
22     .size main,64
```

Listing 7.14: WCC translation of `structPointer.c`

simple `for`-loop. This implementation, however, highlights the usage of pointers in a *relational expression*. Line 13 then writes the value 42 to the member `x` of the `struct foo` currently pointed to by `p`. After the assignment, `p` is incremented to point to the next member of array `fa`.

Listing 7.14 shows the WCC's translation of the small testfile. The two `struct foo*` variables `p` and `q` are initialized in lines 6 and 7 respectively. Since the offsets given in the source file are constant, the *final byte offset* is evaluated at compile-time and both variables, represented by registers `r0` and `r2`, can be initialized with a single instruction.

The realization of the *relational expression* controlling the `for`-loop can be seen in line 10. As was explained in section 6.5.9, the rules implementing *relational expressions* for integers are simply reused here. In this case, it is as simple as a single *comparison* [`CMP`] instruction comparing the two pointers stored in `r0` and `r2`.

Line 13 of `structPointer.c` can be found in lines 13 through 16 in the WCC's assembly translation. Line 13 moves the *source value* `#42` into register `r1` since immediate values cannot be stored directly. Lines 14 and 15 realize the *post-index increment* on `p` (`p++`) by first copying the old address value into r3 and then increasing the value of `p`, represented by `r0`, by 128 bytes, the size of `struct foo`. The old address is then used in line 16 where the actual *store* [`STR`]

Code Selector	**ACET**	**WCET**	**Code Size (Bytes)**
WCC	5599	5610	64
`gcc`	4963	4974	60
Delta in %	13%	13%	7%

Table 7.6.: Evaluation results for `structPointer.c`

takes place. The base address is offset by *124 bytes* to access the integer member `x` that resides behind the 123 bytes for the `char`-array `s` plus the one byte left empty for alignment purposes.

Note that *modification* and *memory access* are not combined into a single instruction here. This is not possible because a single member of the `struct` is overwritten. Since the offset used to access member `x` (124 bytes) is different from the offset that has to be applied to increment pointer `p` (128 bytes), they cannot be combined into a single instruction.

Listing 7.13 lists the `gcc`'s translation for the testfile. The initialization of pointer `p` is realized in lines 6 and 7 of the `gcc`'s translation, split up into two instructions since the *frame pointer* [`fp`]-relative immediate offset is too large to be encoded in a single *subtraction* [`SUB`] instruction. The initialization of `q`, however, has been moved to line 12, similarly to how the `int* q` from `arrayPointer.c` discussed in the previous subsection was also moved in the `gcc`'s translation.

When realizing the assignment of line 13 of the benchmark's source, the `gcc` implements the *post-indexing* in a more efficient way. Instead of moving the old value of pointer `p` into a temporary register, the `gcc` simply performs the store before the increment. In line 9 the value to store (`#42`) is loaded into register `r3` for the same reason outlined earlier. Already, the store is performed in line 10, writing the value to the old address register `p` offset by 124 bytes, where `int`-member `x` resides. Afterwards, the `gcc` performs the increment on register `r2` which represents pointer `p` in line 11.

The `gcc` once again applied the *loop inversion* optimization, transforming the `while`-loop of the C source into a *do-while* loop. As was the case in the previous benchmarks, this causes the `gcc`'s translation to emit one less *branch* [`B`] which is still present in line 17 of the WCC's translation.

Table 7.6 outlines the evaluation results of the `structPointer.c` benchmark. While the instruction count is almost identical (they differ by a single instruction), there is a small but noticeable timing difference. The `while`-statement's loop body is the key contributor to the program's runtime since it is executed 100 times. The loop body of the WCC's translation contains *seven* instructions, whereas the `gcc`'s only consists of *six*. Specifically, the WCC uses two *branches*, one conditional and one unconditional, whereas the `gcc` only uses a single *unconditional branch*.

8. Conclusion and Outlook

The goal of this bachelor thesis was to design and test a code selector ruleset translating *jump* and `switch` *statements* as well as *pointer expressions* from ANSI-C to valid ARMv4 assembly.

Rules for each of the above areas were developed in chapters 5 and 6. When evaluating the developed ruleset under the criterion of *correctness*, the *testbench* testfiles outlined in section 7.1 indicate that the designed rules behave as intended. Of course, the testfiles do not give a guarantee for correctness or completeness, but rather provide a certain level of security that the ruleset generates semantically correct code.

The ruleset developed for *jump statements* is very straightforward and served as an excellent introduction to the topic. Inserting single *unconditional branches* [`B`] is a good solution that a code selector rule, *by itself*, could accomplish. The benchmark evaluated in subsection 7.2.2 made clear that more efficient implementations using, for instance, conditional execution are possible. Such solutions require a more global view of the source than that provided by the **ICD-CG**, however, and are best left to *optimizations*.

The ruleset developed for `switch` statements is functional, but does not exhibit the best performance. Even when excluding the use of *branch tables* for the reasons outlined in subsection 7.2.3, the `gcc`'s benchmark translation made clear that the collection of *comparisons* and *branches* can be organized in a *tree*, realizing the `switch` statement with logarithmic rather than linear time. For `switch` statements with few `case` labels the inserted code performs decently, however, as indicated by the benchmark results given in table 7.4.

The ruleset tackling *pointer expressions* is the most complex of the three and was therefore sectioned off into its own chapter. This complexity is on the one hand inherent to pointers: With *indirection* and *address* operators, *pointer arithmetics* and the interplay between *pointers* and *composed types* it is the most involved of the three areas. But the ruleset's complexity is also the consequence of choice, because the developed ruleset goes out of its way to insert instructions combining *modifications* and *memory accesses* wherever possible, potentially performing both with a single instruction. And as the evaluation in subsections 7.2.4 and 7.2.5 point out, this choice paid off. The manipulation of addresses, combining *modifications* and *memory accesses* and using *bit shifts* whenever applicable led to evaluation results in *runtime* and *code size* on par with those of the `gcc`'s code selector.

Despite the advances made in this thesis, a lot of work remains to be done on the WCC's internal code selector. There are still many expressions currently unimplemented, such as bit-shifts (`<<`), the modulo-operator (`%`) or division (`/`). Moreover, support for types wider than 32 bits, such as `double` or `long long`, remains open as well. Finally, there are also more advanced C features such as *function pointers* or *bit-fields* in `struct` types that still have to implemented.

A. CD Contents

Directory	Contents
`Bachelor_Thesis_TobiasMarschner_2019.pdf`	This document provided in the *Portable Document Format* (PDF).
`ruleset/`	The source code of the developed ruleset.
`testbench/`	The 126 *testbench* testfiles.
`evaluation/`	The four testfiles used in section 7.2 including the WCC and `gcc` compilations.

Table A.1.: Directory Structure of the accompanying CD

Table A.1 printed above provides an overview of the files included on the accompanying CD. Next to this very document the source of the developed ruleset from chapters 5 and 6 as well as the *testbench* and *evaluation* testfiles from chapter 7 are included.

List of Figures

List of Tables

List of Listings

Bibliography

[1] Peter Marwedel. *Embedded System Design : Embedded Systems Foundations of Cyber-Physical Systems*. Dordrecht: Springer Science & Business Media, 2011.

[2] Edward A. Lee. "The Future of Embedded Software". In: ARTEMIS Conference. Graz, Austria, 2006.

[3] Edward A Lee. *Computing Foundations and Practice for Cyber-Physical Systems: A Preliminary Report*. Tech. rep. UCB/EECS-2007-72. University of California, Berkeley, 2007.

[4] Hermann Kopetz. *Design Principles for Distributed Embedded Applications*. Dordrecht: Springer Science & Business Media, 2011.

[5] Paul Lokuciejewski and Peter Marwedel. *Worst-Case Execution Time Aware Compilation Techniques for Real-Time Systems*. Dordrecht: Springer Science & Business Media, 2011.

[6] J. A. Stankovic. "Misconceptions about real-time computing: a serious problem for next-generation systems". In: *Computer* 21.10 (1988), pp. 10–19.

[7] Dominic Paul Oehlert, Arno Luppold, and Heiko Falk. *Compilation for Real-Time Systems : An Overview of the WCET-aware C Compiler WCC*. Deutsche Forschungsgemeinschaft (DFG), 2018.

[8] Abir Bouraffa. "WCC Code Selection Framework for the ARM Processor Architecture". Project Work. Hamburg University of Technology, 2017.

[9] Janina Plog. "Developing a Code Selector's Ruleset for the ARM Processor". Project Thesis. Hamburg University of Technology, 2017.

[10] Paavo Becker. "Development of a Code Selector Rule Set for Composed Types for the ARM Architecture inside a WCET-aware Compiler". Bachelor Thesis. Hamburg University of Technology, 2018.

[11] Jan Runge. "Design of a Code Selector Rule Set for Type Casting for ARM Processors as a Part of a WCET-Aware Compiler". Bachelor Thesis. Hamburg University of Technology, 2018.

[12] ISO/IEC 9899:1999. *Programming languages – C*. Standard. Geneva, CH: International Organization for Standardization, 1999.

[13] Alfred V. Aho et al. *Compilers : Principles, Techniques, and Tools*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[14] Raimund Kirner. “Extending Optimising Compilation to Support Worst-Case Execution Time Analysis”. PhD thesis. Wien University of Technology, 2003.

[15] AbsInt Angewandte Informatik GmbH. *aiT Worst-Case Execution Time Analyzers*. Accessed: 25.10.2019. URL: `https://www.absint.com/ait/index.htm`.

[16] Synopsys Inc. *Synopsys CoMET-METeor*. 2018.

[17] Timon Kelter and Peter Marwedel. “Parallelism analysis: Precise WCET values for complex multi-core systems”. In: *Science of Computer Programming* 133 (2017), pp. 175–193.

[18] *ICD-C Compiler framework*. Accessed: 16.10.2019. URL: `www.icd.de/de/eingebettete-systeme/icd-c-compiler/icd-c`.

[19] *ICD-LLIR Low-Level Intermediate Representation*. Accessed: 16.10.2019. URL: `https://www.icd.de/de/eingebettete-systeme/icd-c-compiler/icd-llir`.

[20] *ARM7TDMI Technical Reference Manual*. Version r4pi. ARM Limited. 2004.

[21] *ARM Architecture Reference Manual*. Version H. ARM Limited. 2005.

[22] Mostafa Abd-El-Barr and Hesham El-Rewini. *Fundamentals of Computer Organization and Architecture*. Hoboken, N.J: Wiley, 2005.

[23] David A. Patterson and John L. Hennessy. *Rechnerorganisation und Rechnerentwurf*. Berlin: De Gruyter Oldenbourg, 2016.

[24] *Procedure Call Standard for the ARM Architecture*. Version IHI 0042F. ARM Limited. 2015.

[25] *ICD-CG Code Generator*. Accessed: 26.10.2019. URL: `http://www.icd.de/de/eingebettete-systeme/icd-c-compiler/icd-cg`.

[26] Alfred V. Aho, Mahadevan Ganapathi, and Steven W. K. Tjiang. “Code Generation Using Tree Matching and Dynamic Programming”. In: *ACM Trans. Program. Lang. Syst.* 11.4 (1989), pp. 491–516.

[27] Christopher W. Fraser, Robert R. Henry, and Todd A. Proebsting. *BURG--Fast optimal instruction selection and tree parsing*. Tech. rep. University of Wisconsin-Madison Department of Computer Sciences, 1991.

[28] Christopher W. Fraser, David R. Hanson, and Todd A. Proebsting. “Engineering a Simple, Efficient Code-generator Generator”. In: *ACM Lett. Program. Lang. Syst.* 1.3 (1992), pp. 213–226.

[29] Steven W. K. Tjiang. *An Olive Twig*. Tech. rep. Synopsys Inc., 1993.

[30] Guido Costa Souza De Araujo. “Code generation algorithms for digital signal processors”. PhD thesis. Princeton University, 1997.

[31] *TriCore V1.6 User Manual*. Version 1.0. Infineon Technologies AG. 2012.

[32] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000.

[33] Daniel Page. *Practical Introduction to Computer Architecture*. Ed. by David Gries and Fred B. Schneider. Dordrecht: Springer Science & Business Media, 2009.