Extending tree pattern matching for application to peephole optimizations

van Oirschot, J.

Award date:
2019

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Department of Electrical Engineering
Electronic Systems Group

Extending tree pattern matching for application to peephole optimizations

Master Thesis

Janek van Oirschot

Supervisors:
Prof. dr. Henk Corporaal
Dr. ir. Roel Jordans
Dr. ir. Loek Cleophas
Nuno Lopes, PhD.

Eindhoven, March 2019

Abstract

Peephole optimizations are one of the many optimizations used by compilers. These peephole optimizations are small static code transformations that occur whenever possible without cost analysis. Because these peephole optimizations are small, many of them are required to make an impact on the execution time of a compiled program. Within the LLVM compiler suite, the peephole optimizer is not only used to optimize but also to canonicalize code snippets for other optimizations. Because of its canonicalizing nature, the peephole optimizer requires even more peephole optimizations and is applied multiple times to restore the canonicalizations that may have been undone by other optimizations. Because of these characteristics, the peephole optimizer in LLVM is one of the dominating factors in the runtime of LLVM's optmization passes.

Alive aims to provide both a Domain Specific Language (DSL) to describe peephole optimizations, and prove the correctness of these peephole optimizations for LLVM. Furthermore, the Alive DSL describes peephole optimizations adequately to automatically generate code which can then be used within LLVM. This generated code, however, still uses the same matching strategy as LLVM's default pattern matching strategy for peephole optimizations.

This thesis explores different tree pattern matching methods, extends them to adhere to constraints introduced by peephole optimizations, and implements code generators within Alive which utilize the augmented tree pattern matchers with the goal to speed up compilation time of LLVM's peephole optimizer. In doing so, it shows that, given a limited set of peephole optimizations, it is possible to speed up compilation time using tree pattern matching compared to LLVM's default matching strategy, without sacrificing execution time or binary size.

Acknowledgements

I would like to take this opportunity to thank my graduation supervisors. Henk Corporaal, for his guidance and candid feedback whenever I needed it. Roel Jordans, for his guidance, advise, and willingness to always listening to my bi-weekly ramblings on my graduation progress. Loek Cleophas, for joining my committee, providing feedback, and advising on the subject matter. And finally, Nuno Lopes, for providing the opportunity to intern at Microsoft Research in Cambridge, UK and always being ready to guide and give feedback whenever I would knock yet another time on his office door.
Again, I thank you all.

Contents

List of Figures

List of Tables

List of Abbreviations

Abbreviations

BFS Breadth First Search

DAG Directed Acyclic Graph

DFA Deterministic Finite Automata

DSL Domain Specific Language

IR Intermediate Representation

LLVM Low-Level Virtual Machine

NFA Non-Deterministic Finite Automata

PF Pattern Forest

SMT Satisfiability Modulo Theories

SSA Static Single Assignment

Chapter 1

Introduction

Today's world is governed by technology. Many, if not all, have integrated it into their daily lives, and businesses depend on it for their processes to the point where technology has become an integral part of our lives. Of course, technology comes in many forms and shapes; from word processors to embedded devices. One common aspect in the majority of technology today is the use of software, usually written by software developers. This software then runs on computers, embedded devices, servers, and much more. However, software written by developers are generally in a human readable programming language which cannot be directly run on devices. The software written in these high-level programming languages still requires translation to a format that can be interpreted by devices. This translation is done by a so called compiler.

The compiler, which translates human readable programs into a format executable by devices, is also a program described in a programming language that has to be translated into a representation runnable on a device. Problems in regular applications result in crashes of said programs; however, problems within the compiler might result in crashes of the programs compiled (i.e. translated). Meaning that the negative effects of compiler bugs transcend the compiler itself, onto the software being compiled. Because of this trait, a compiler's correctness ought to be ensured as well as possible.

Not only should the compiler ensure that the translation does not result in faulty emitted executable programs, it should also ensure a speedy runtime such that emitted programs run as efficiently as possible. To ensure a feasible runtime for emitted programs, many optimizations take place within the compiler. These focus on reducing a program's runtime or, alternatively, reducing an emitted program's size. Supporting all of these optimizations within the compiler means that the software describing the compiler itself is enormous.

One technique to allow easier maintainability and adaptability of the compiler software is to abstract parts of the code and lift them into a new Domain Specific Language (DSL), accompanied with a program which takes the DSL and emits part of the compiler's software. An example of this would be within the LLVM compiler suite where the TableGen[1] DSL is used to describe device specific characteristics. Another example of a DSL, utilized in compilers, is within the Go compiler[2] where a DSL is used to describe peephole optimizations (pattern match code rewriting) which can be translated to the Go language (the programming language used by the Go compiler).

Alive [9] aims to be a DSL and verifier for peephole optimizations within the LLVM compiler suite which focusses on proving the correctness of the peephole optimizations.

[1] https://llvm.org/docs/TableGen/index.html
[2] https://github.com/golang/go

Proving correctness of peephole optimizations decreases the flaws within LLVM, resulting in less erroneous executable programs emitted by LLVM. Additionally, Alive describes the peephole optimizations sufficiently to generate part of the LLVM peephole optimization code in C++, the primary programming language used within the LLVM project. Both verification and code generation allow developers working on LLVM to easily adapt LLVM's peephole optimizer for new peephole optimizations, and prove the correctness of these extensions.

The current C++ code generator used by Alive emits code in the same structure as the manually maintained peephole optimizer in LLVM. This means that the code emitted is still readable as maintaining and extending peephole optimizations manually requires readable code to aid the developers. However, it is possible to sacrifice the human readable aspects of the peephole optimizer's C++ code if, in turn, it can speed up compilation time. Moreover, maintaining and extending the peephole optimizer then occurs on a higher level, on the Alive DSL as the C++ code is then generated using the Alive DSL. This is similar to how a software program loses its abstractions after compilation to an executable program for the ability to run the program efficiently on a particular device.

This thesis explores alternative methodologies which trades off clarity of the peephole optimization C++ code for compilation speed. More specifically, alternate peephole optimization pattern matching techniques are explored where the Alive DSL definitions of peephole optimizations are translated into C++ code utilizing said alternative methodologies. One of the strategies explored is called top-down pattern matching, denoting the direction in which matching occurs. The other strategy explored is called bottom-up, also named after the direction that pattern matching occurs.

The goal of the project is to automatically generate C++ code for the peephole optimizer (using Alive), ready for use within LLVM where emitted code uses tree pattern matching methodologies which are modified to conform the constraints introduced by peephole optimizations, and to speed up execution time of the peephole optimizer within LLVM. Section 3.2 explains these goals more in depth.

This thesis achieves the following:

- Describe additional constraints for tree pattern matching required to support peephole optimizations
- Extend existing top-down and bottom-up tree pattern matching construction algorithms to conform to peephole optimization constraints
- Implement an initial version of both top-down and bottom-up tree pattern matchers, which support peephole optimizations, in Alive to automatically generate necessary C++ code for LLVM's peephole optimizer
- Evaluate the initial implementations with LLVM's default peephole optimizer and Alive's default peephole optimizer generator

1.1 Overview

Before describing the project and its problems, the required background knowledge is introduced in chapter 2. After that, the chapter 3 explains the motivation and problem statement of the project. Then, the extensions required for both top-down and bottom-up tree pattern matching stategies to accommodate constraints introduced by

peephole optimizations in LLVM are introduced in chapter 4 together with a description of the initial implementations of both strategies. These implementations are then benchmarked where the results of these benchmarks are described in chapter 5. Finally, a conclusion summarizing the work with current issues and recommendations on future work is described in chapter 6.

Chapter 2

Preliminaries

This chapters aims to provide sufficient preliminary knowledge for the remainder of this thesis. Subjects covered in this chapter will be:

- Low-Level Virtual Machine (LLVM), its design and characteristics
- Peephole optimizers, and LLVM's peephole optimizers' design
- Alive, a tool to automatically formally prove correctness of peephole optimizations within LLVM
- Finite automata, their structure, and execution
- Tree pattern matching, its concept and relevant research

2.1 LLVM

The LLVM project is an open source project[1] that supports and maintains a set of modular libraries most suitable for compilers. Just the LLVM project itself would not suffice for a complete compiler toolchain as it only consists of separate libraries. As a subproject of LLVM, the clang project exists which can be considered the front-end that uses the LLVM libraries for its back-end. What sets it apart from other open source compiler toolchains is its modularity and its adaptability without sacrificing compiler performance [7]. Its modularity refers to the reusability of the compiler submodules. Its adaptability refers to the ease of altering a submodule as the software design of LLVM took separation of concerns into account. An example of how both adaptability and modularity plays a role in LLVM's software design is the human readable LLVM Intermediate Representation (IR). This LLVM IR can be considered a high-level assembly language that can be converted to more efficient LLVM IR, or alternatively, converted to (a computer architecture's) target specific assembly. Support for new programming languages could be implemented more easily by creating a front-end that emits LLVM IR for said new programming language. Of course, this is easier said than done but can still be considered easier in comparison to other compiler toolchains which have a tighter integration of its submodules. A front-end developer should not need to know back-end details just to develop a new programming language. A few examples of programming languages that have front-ends that are built upon LLVM are: Rust, Swift, and Halide. These programming languages depend upon LLVM's optimizer and back-end for functionality while they maintain a custom front-end.

[1] llvm github mirror: https://github.com/llvm-mirror/llvm

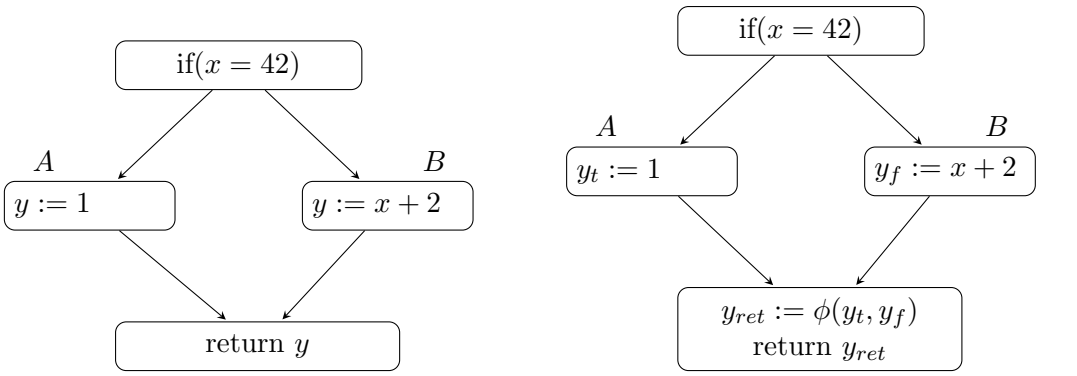
2.1.1 LLVM design

This section aims to explain the LLVM's background and its design. Namely, it will describe LLVM's IR, some ideas behind it and how it relates to the LLVM compiler's design.

LLVM Intermediate Representation

LLVM's IR is the primary language used within LLVM in terms of code representation. More traditionally, a compiler's IR is internally described in software and data structures. However, LLVM's IR can also be emitted in human readable format (shown later in this chapter). The appeal of such a design is that code transformations can be tracked as code goes through the compiler pipeline allowing for easier understanding of the compilation (and optimization) process.

LLVM's IR appears similar to a verbose, high-level assembly language. However, some details are abstracted away such as the constraint of a finite amount of registers. Instead, an infinite amount of virtual registers is assumed. Moreover, all virtual registers are assigned only once during their lifetime. This type of assignment is also called Static Single Assignment (SSA). The strength of SSA is that it simplifies analysis of the assign-use relation and the lifetime of values. Altogether, this allows for easier data and control flow analysis [1] within the compiler.



(a) *Joining basic blocks* ***(b)*** *Joining basic blocks with SSA form*

Figure 2.1: *Multiple branch joining example*

Sometimes, value assignment of variables can be ambiguous due to the joining of basic blocks, as is shown in figure 2.1a. Here, multiple blocks are shown, each with a single entry point and a single exit point i.e. basic blocks where the only entry point is at the start, and only exit point is at the end of the code sequence with no way to branch during the sequence [3]. In such a case, assignment of a variable occurs multiple times. Namely, in basic block A, and basic block B. As multiple assignments of the same variable is not possible in SSA, a so called phi node is introduced (ϕ) which assigns values to variables depending on which basic block branched to the phi node's basic block. The SSA equivalent with such a phi node is shown in figure 2.1b. Here, the same basic blocks can be observed as in figure 2.1a; however, the variables within the SSA form equivalent basic blocks of figure 2.1b differ. As can be seen, basic block A now assigns a value to variable y_t whereas basic block B now assigns a value to variable y_f. Moreover, the basic block where both basic blocks A and B join now assigns a value to variable y_{ret} using the phi node. The first argument of $\phi(y_t, y_f)$ denotes that if the branch originates from

A, y_t should be used for assignment. The second argument denotes that y_f should be used for assignment if the branch originates from B.

```
int f(int a, int b) {
  while (b < a) {
    b++;
  }
  return b;
}
```

Listing 2.1: *C code example with multiple joining basic blocks*

```
define i32 @f(i32 %a, i32 %b) {
entry:
  br label %cond

cond:
  %local = phi i32 [ %b, %entry ], ↩
      [ %inc, %body ]
  %cmp = icmp slt i32 %local, %a
  br i1 %cmp, label %body, label ↩
      %end

body:
  %inc = add nsw i32 %local, 1
  br label %cond

end:
  ret i32 %local
}
```

Listing 2.2: *LLVM IR example with multiple joining basic blocks*

Listing 2.2 shows an example of LLVM IR's SSA form for the equivalent C code in listing 2.1. In listing 2.2, the 4 basic blocks observed in LLVM IR are 'entry', 'cond', 'body', and 'end'. As can be observed, the 'cond' basic block contains a phi node where the origin is either the 'entry' basic block, or the 'body' basic block. If the 'entry' basic block is the origin, then the value in $\%b$ will be assigned to $\%local$, if the 'body' basic block is the origin, then the value in $\%inc$ will be used instead.

LLVM IR Passes

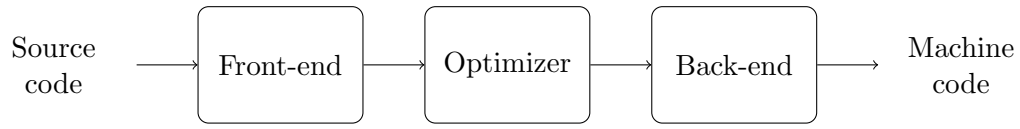


Figure 2.2: *Compiler design*

Figure 2.2 shows the traditional design of a compiler. The design of LLVM does not deviate much from this design. In LLVM, the front-end (clang) will convert source code to LLVM IR. This LLVM IR from the front-end will then serve as the input for the optimizer which contains a plethora of optimizations and analyses. After finishing these optimizations, more efficient LLVM IR will be emitted. This optimized LLVM IR will then be forwarded to the back-end which will use it to emit corresponding machine code depending on what computer architecture is being compiled for.

Notice that both the input and output of the optimizer is in LLVM IR format. To some extent, it is possible to skip optimizations altogether and immediately go from front-end to back-end. This will, however, emit less efficient machine code. Within LLVM, the design of the optimizer with its optimizations has been implemented in so called passes. A pass iterates over (part of) the LLVM IR and either analyses or transforms it. These passes can be configured and extended depending on the particular

needs of computer architectures.

2.1.2 Peephole optimizer

Peephole optimizations are optimizations applied on small, typically three lines of LLVM IR, snippets of code. Peephole optimizations are applied wherever possible, meaning that if a match has been found which adheres to the required preconditions the transformation is applied. If a snippet of code matches one of the peephole optimizations, it will be transformed into a more efficient alternative.

LLVM, much like many other compiler toolchains, incorporates a peephole optimizer. This peephole optimizer in LLVM runs during LLVM's optimization phase. Within LLVM these peephole optimizations are implemented using passes. The peephole optimizer passes in LLVM are the instcombine (instruction combine), and instsimplify (instruction simplify) passes. An example of instruction simplification is the reduction of same variable subtraction to a 0 i.e. $x - x \rightarrow 0$ which simplifies the expression and might enable other optimizations. An example of instruction combination is canonicalization of code with multiplication by a power of 2 to a left-shift i.e. $x \times 2 \rightarrow x \ll 1$ whose reduction to a shift does not revert back to a multiplication.

Algorithm 1 shows pseudocode for naive instcombine's algorithm in LLVM. The InstructionCombine function runs over each function within the program that's being compiled. For every function being compiled, the sequence of LLVM IR instructions within said function are copied (in the same order as they occur in *func*) to an auxiliary sequence wl on line 3, which is iterated over until wl is empty. Each iteration of the loop starting at line 5 starts with splitting the sequence into the sequence consisting of only the front instruction i, and the sequence with remaining instructions wl'. Then wl is updated by assigning remaining instruction sequence wl' to wl. This operation mimics a stack's 'pop' operation where only the top is removed from a stack. After retrieval of front instruction i, it is put through the 'optimize' function on line 8 which attempts to optimize i and all instructions necessary for i's execution (i.e. 'optimize' might affect instructions other than i, albeit indirectly). The 'optimize' not only tries peephole optimizations but also checks whether i should be removed if it produces an unused value (Dead Code Elimination). The 'optimize' function returns a pair (i', j) where i' is the new instruction or value to replace i with, and j is the sequence of instructions that need to be added with i' to ensure correctness. Whenever $i' = i$, no optimizations were found, hence the replacement condition on line 9 shows that replacements only occur when $i' \neq i$. Replacements of instructions in the $func$ sequence occurs on line 10 where the 'replace' function replaces i with the sequence of instructions $j \cdot [i']$ which is the concatenated instruction sequence of j and i'. Note that after running this replace on line 10 i does not occur in $func$ anymore and has been deleted. After replacement, the auxiliary instruction sequence wl needs to be updated with instructions to iterate over again. First, the sequence j is added to the front of the wl sequence, then all users of i' not in wl yet, are added to the front. Note that the users of i' are all instructions that need i''s value. After adding all user instructions of i', i' is added to the front. After all added instruction sequences (line 11-15) to wl', the sequence wl should look as follows: $[i'] \cdot users(i') \cdot j \cdot wl'$, where left of the sequence is the front, and right of the sequence is the back. Keep in mind that if any changes occurred to $func$, the *changed* boolean will be set and the loop on line 2 will repeat all aforementioned operations until *changed* won't be set after going through $func$ without apply any changes.

The algorithm does not have an upper-bound on runtime complexity as fixed-point

Algorithm 1 Naive instcombine

input:

$func : [inst]$	-	sequence of scheduled, reachable instructions

vars:

$wl : [inst]$	-	sequence of instructions
$changed : \mathbb{B}$	-	whether code was changed
$i : inst$	-	instruction to be optimized
$i' : inst$	-	optimized instruction
$u : inst$	-	user instruction
$j : [inst]$	-	sequence of instructions
$optimize : inst \to inst \times [inst]$	-	DCE, constfolding, and peephole optimizations
$users : inst \to [inst]$	-	returns sequence of user instructions of instruction sequence
$replace : [inst] \times inst \times [inst] \to [inst]$	-	replaces instruction found in sequence with replacement sequence

1: **function** InstructionCombine($func$)
2: **repeat**
3: $wl := func$
4: $changed := false$
5: **while** $wl \neq []$ **do**
6: $[i] \cdot wl' = wl$
7: $wl := wl'$
8: $(i', j) := optimize(i)$
9: **if** $i' \neq i$ **then**
10: $func := replace(func, i, j \cdot [i'])$
11: $wl := j \cdot wl$
12: **for all** $u \in users(i')$ **do**
13: **if** $u \notin wl$ **then**
14: $wl := [u] \cdot wl$
15: $wl := [i'] \cdot wl$
16: $changed := true$
17: **until** $changed = false$

$$optimize(i) = \begin{cases} \text{(i', j),} & \text{if i' is the new instruction, and sequence j needs to be added to func} \\ \text{(i, []),} & \text{if no change needs to be applied} \\ \text{(nil, []),} & \text{if i should be deleted} \end{cases}$$

rewriting iterations have the potential to run without terminating if circular rewriting rules exist. However, the lower-bound on runtime complexity is $\Omega(n)$, where n is the amount of LLVM IR instructions (i.e. only one iteration needed). Ideally, the runtime is decreased without sacrificing performance of the compiled program.

The pattern matching procedure occurs within the 'optimize' function. This function will take the input instruction i as root of the expressions to match. This matching is done within a sequence of conditionals which explicitly check the shape of the tree rooted at i and its preconditions. This results in a linear scan for a matching peephole optimization. Moreover, the priority of these patterns are dictated by their order of occurrence as the linear scan matches the first conditional that holds. The resulting runtime complexity associated with 'optimize' will be $O(k \times m)$, where k is the number of patterns to match and m is the pattern size in number of tree nodes.

2.1.3 Alive

Alive is a DSL and tool for formal verification of peephole optimizations [9]. Its verification considers LLVM's constraints and is therefore primarily aimed at LLVM. Alive verifies the transformations of a peephole optimization. It does not automatically generate any peephole optimizations but assumes the peephole optimizations are either already written by a developer or automatically generated by other tools. Verification is done using Satisfiability Modulo Theories (SMT) where it either proves a peephole optimization is correct or constructs a counter-example in which the proposed peephole optimization does not hold.

```
Pre: width(%x) > 1
%r = mul %x, 2
     =>
%r = shl %x, 1
```

Listing 2.3: *Alive DSL example*

As Alive is aimed at peephole optimizations for LLVM, the Alive DSL shares similarities with LLVM IR. Shown in listing 2.3 is an example of Alive's DSL. This example shows what Alive has to verify, namely that the snippet of code that occurs before the arrow $\Rightarrow$ is allowed to be transformed into the snippet of code that occurs after the arrow $\Rightarrow$, given that the precondition which says that the bit width of $\%x$ is greater than 1 holds.

The syntactic similarities can be observed by comparing Alive's DSL, as shown in listing 2.3, to LLVM IR as shown in listing 2.2. A few differences to LLVM's IR are:

1. The addition of a precondition. This strengthens the transformation and constraints possible scenarios.

2. Removal of data type. The type of a variable can be omitted as either the peephole optimization is not type-specific, or the type can be inferred from the transformation.

3. Introduction of variable constants. Constants and variables differ, however, constants must still be abstracted in Alive. A transformation for each and every possible constant value is not feasible therefore constants are abstracted away using variable constants. These can be considered constrained variables as the variables are constrained to constant values only.

4. Abstract rewriting-like notation. All Alive transformations have a structure similar to a rewrite rule within an abstract rewriting system. Alive's transformations follow the following form: $A \Rightarrow B$ where A is unoptimized code and B is (supposed to be) its optimized equivalent [9].

Transformations in Alive are not checked for performance, merely for validity of transformation.

Alive built-in functions and predicates

Alive supports a set of built-in predicates, for preconditions, and a set of built-in functions, for expressions. These built-in functions and predicates can be used to aid in constraining expressions if necessary. For example, listing 2.3 shows one of the available built-in functions in Alive: the 'width' function used to retrieve the bit width of an expression's value. In this example it is used to constrain the bit width of subexpression 'x' to be wider than 1 bit.

Code generation

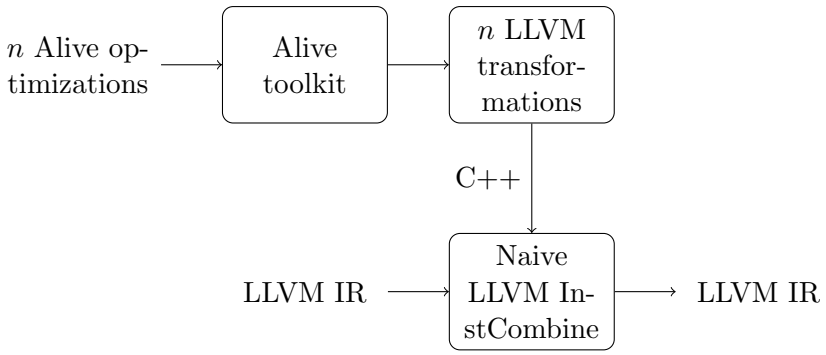


Figure 2.3: *Alive InstCombine*

In addition to validating LLVM peephole optimizations, Alive can also automatically generate the C++ code of the peephole optimizations that can be inserted into LLVM. The advantage of automatic generation is that the peephole optimizations will be both proven correct, and no time will be lost implementing the C++ equivalent. Currently, however, emitted C++ code still is implemented using a naive InstCombine methodology as shown in section 2.1.2.

Figure 2.3 shows an overview of how Alive and LLVM's InstCombine interact. A number of peephole optimizations described in Alive's DSL are provided to the Alive toolkit. This toolkit can then automatically generate the equivalent C++ code for said peephole optimizations. This generated C++ code is ready to be included by LLVM's InstCombine and then compiled. The LLVM tools that use InstCombine will then apply only the peephole optimizations described in the original Alive DSL.

2.2 Finite Automata

The tree pattern matching problem described in chapter 4 uses automata to solve tree pattern matching. The precomputations in tree pattern matching are, more often than not, automata that are created for a given set of tree patterns, without the need of

an input tree to match on. This is very similar to the Aho-Corasick string matching algorithm that precomputes the search strings' finite automaton without the need for an input string [2]. The advantage of using automata for computational problems is that it is well-established within theoretical computer science. Many (optimization) algorithms exist for automata, which do not depend on the context of said automata.

This section aims to be an introduction to automata, starting with the general idea of finite automata.

2.2.1 General Finite Automata

Finite automata are abstract machines with a finite number of states, and transitions to other states, used to solve computational problems [3]. When an automaton runs it is always in a state where it can, depending on the current input symbol, transition to another state until no more input symbols exist. Prior to processing input symbols, an automaton needs a start state (also called initial state) from where it starts executing. The final or accepting states are the states that accept the input depending on the computational problem being solved. For example, to verify whether an input string adheres to a particular pattern, an automaton can be constructed for the string pattern. This automaton will accept an input string if there exists a path, using the whole input string, that reaches a final state.

2.2.2 Deterministic Finite Automata

Deterministic Finite Automata (DFA) are automata with deterministic properties. More formally, a DFA is defined as follows:

Definition 2.2.1. A Deterministic Finite Automaton (DFA) is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where

- Q is the finite non-empty set of states
- Σ is the input alphabet, a finite set of input symbols
- δ is the transition function $Q \times \Sigma \to Q$
- $q_0 \in Q$ is the initial or starting state
- $F \subseteq Q$ is the set of final or accepting states

Note that every DFA is also a Nondeterministic Finite Automaton (NFA) with some constraints put into place to enforce its determinism. These constraints are:

1. ϵ transitions, which are transitions that may occur without consuming an input symbol from the input alphabet Σ, are not allowed. Introducing an ϵ transition into a DFA immediately renders it an NFA.
2. Only a single destination state exists for each unique δ input pairs. This means that for all δ input pairs in $(Q \times \Sigma)$, there can be at most one destination state. Similar to restriction 1, introduction of multiple destinations converts the DFA to NFA.

As can be observed from the constraints, a DFA is merely a restricted NFA. By introducing said restrictions to a NFA (e.g. using Thompson's algorithm [14]), it can be

converted to its DFA equivalent, meaning that if there is an NFA accepting a pattern $p \in \Sigma^*$, then there exists a DFA which accepts that same pattern p [6].

For tree pattern matching, however, an augmented notion of DFAs are used. Similar to regular expression matching, an automaton is constructed given a description of the tree patterns to be matched. Then, a tree is supplied to said automaton to see if it can be matched to any of the patterns. How the DFA ought to be modified will be explained after introducing DFAs.

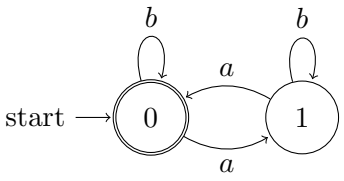


Figure 2.4: *Deterministic finite automaton D*

To illustrate, take $Q_D = \{0, 1\}$, $\Sigma_D = \{a, b\}$, $\delta_D : \{(0, a) \to 1, (0, b) \to 0, (1, a) \to 0, (1, b) \to 1, \}$, and $F_D = \{0\}$.

Then, DFA $D = (Q_D, \Sigma_D, \delta_D, 0, F_D)$ is illustrated in figure 2.4. This figure shows that acceptance occurs on no input symbol, or an even number of a symbols (where b symbols are allowed to occur between the a symbols). Any input string with an odd number of a symbols will be rejected as an odd number will always end up in state 1 which is not a state in the final set.

DFAs can be constructed from NFAs, however, to accommodate the restrictions more states might be necessary [15]. Minimization techniques for DFAs exist but will result in more time spent on construction [3, 4]. Simulation on DFAs, however, can be done in linear time with respect to the input string. As there is no non-determinism in a DFA, there exists only a single path in the DFA for each unique input string.

DFA modification for (top-down) tree pattern matching

As previously described, tree pattern matching uses augmented DFAs for its matching. The reason for such modification compared to a regular DFA is because tree pattern matching can be supplied with multiple tree patterns to match which all have to be accounted for in matching, unlike regular expression matching which only matches a single regular expression. For the DFA this means that regular expression matching does not need to bind a context to a particular final state in $F \subseteq Q$. A match for any state $s \in F$ denotes a match on the one regular expression being matching.

Tree pattern matching, however, requires each final state $s \in F$ to contain additional context on which tree pattern has been matched. The idea that a DFA results in a 'yes' or 'no' given a problem (like regular expression matching) changes. Not only does the tree pattern matching automaton return a 'yes' or 'no' to the problem whether a tree pattern match has been found given an input tree, it also requires the context of which tree pattern has been matched. Because more context is necessary for final states, metadata needs to be added during construction. Moreover, some automata algorithms such as minimization need to be augmented as these assume that all final states are independent of context and can be merged into a single final state.

2.3 Tree pattern matching

Strategies exist to extend pattern matching for trees. With tree pattern matching, a tree pattern is supplied which is matched onto a subject tree. This is similar to how a regular expression describes a pattern which can match a particular input string.

This section aims to introduce the background knowledge necessary for tree pattern matching accompanied with different tree pattern matching methodologies.

2.3.1 Tree pattern preliminaries

An expression (denoted $\mathcal{E}$) is a recursively constructed formula of variable symbols $\mathcal{V}$ and function symbols from an alphabet Σ with each symbol a finite arity[2] (i.e. number of arguments) where an arity of 0 dictates a constant value. For example, take $\Sigma = \{a, b, c\}$, $x, y, z \in \mathcal{V}$, and expression $t = a(b(x, c), y, z)$, then the arity of a is 3, the arity of b is 2, and the arity of c is 0. The subexpression $b(x, c)$ in t is still considered an expression, hence expressions can be recursively constructed. More formally:

Definition 2.3.1. An expression is either:

1. a variable from the set of variables $\mathcal{V}$, or
2. a nullary expression, i.e. a constant, or
3. an n-ary expression where $n > 0$ and each of the n arguments is an expression

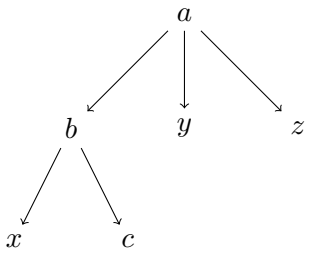


Figure 2.5: *Tree of expression t*

An expression can be considered a tree where each descent from an expression's function symbol to one of its arguments is a descent from a tree node to (one of) its child node(s). For example, the tree of t in the aforementioned paragraph is depicted in figure 2.5. The terminology tree and expressions will be used interchangeably as both express the same concept.

Definition 2.3.2. A path of an expression is:

1. the empty string Λ, denoting the root of an expression, or
2. $p.i$, where p is a path and i is an integer denoting the i^{th} argument for the symbol at path p, of an expression

Given a path p and expression e, to get a subexpression of e at position p (denoted $e|_p$) take p's string and use it to descend the expression e starting at its root. For example, take $e = sub(a, add(mul(b, c), d))$, then $e|_1 = a$, and $e|_{2.1} = mul(b, c)$.

[2] in other literature, also referred to as rank

Furthermore, given a path p and expressions e and e', a replacement of a subexpression in e at position p with expression e' (denoted $e_p[e']$) is done by inserting e' at $e|_p$. For example, take $e = add(add(a, sub(b, c)), d)$, then $e_{1.2.2}[mul(x, y)] = add(add(a, sub(b, mul(x, y))), d)$, and $e_1[z] = add(z, d)$.

The size of an expression e (denoted $|e|$) is the number of nodes (i.e. number of subexpressions including ones of $arity = 0$ and variables $\mathcal{V}$) within e. For example, if $e = add(a, b)$ then $|e| = 3$, and if $e' = sub(add(a, b), sub(c, mul(d, e)))$, then $|e'| = 9$.

Substitutions σ are mappings from variables to expressions. Take $\sigma = \{x \to a, y \to b(x, a), z \to v\}$ where $\Sigma = \{a, b, f\}$, and $x, y, z, v \in \mathcal{V}$, then $\sigma(f(f(x, a), f(y, z)) = f(f(a, a), f(b(x, a), v))$.

Definition 2.3.3. An expression a is said to be an instance (also referred to as subsumption i.e. b subsumes a) of expression b (denoted $a \leq b$, or equivalently $b \geq a$) if there exists a substitution σ for variables in b such that a can be constructed. For example, $f(g(x, y), z)$ is an instance of $f(x, y)$ given the substitution $\sigma = \{x \to g(x, y)\}$. Moreover, an expression b strictly subsumes a ($b > a$) if $b \geq a$ and $b \neq a$.

Definition 2.3.4. A distinction is made between a subject tree and pattern tree. Pattern trees are expressions that describe the search query whereas subject trees are expressions that describe the tree queried on. Additionally, pattern trees may have wildcards i.e. variables. The variables in a pattern tree denote any sub-tree, hence other literature refers to these as wildcards. When multiple of the same wildcard variables occur in a pattern, the subtrees at these wildcard variables ought to be equal, unless specified otherwise.

A pattern forest (PF) of a set of patterns P is the set of all possible subtrees of all pattern trees in P, including the patterns trees themselves. To illustrate, let $P = \{a(a(b, x), b), a(a(x, c), c)\}$, then $PF = \{a(a(b, x), b), a(a(x, c), c), a(b, x), a(x, c), b, c, x\}$.

Definition 2.3.5. The properties used to construct a match set:

1. If expression a is a nullary symbol, then $match(a) = PF \cap \{a, x\}$ where x denotes a wildcard variable

2. If expression a is a n-ary symbol with $n > 0$, then $match(a(t_1, ..., t_n)) = PF \cap (\{x\} \cup \{p \mid p \text{ has root } a, \text{ and } 1 \leq j \leq n, child_j(p) \text{ is in } match(t_j)\})$ where x denotes a wildcard variable and $child_j(p)$ is the j^{th} subexpression of expression p

Informally, all match sets denote all possible matching scenarios of the subtrees in the pattern forest. For example, take expressions $P = \{a(a(b, x), b), a(a(y, c), c)\}$ where $\Sigma = \{a, b, c\}$ and x, y are wildcard variables. Then, $match(a(b, c)) = \{a(b, x), a(x, c), x\}$, and $match(a(b, a(c, b)) = \{a(b, x), x\}$.

Definition 2.3.6. A pattern p matches a subject tree t if:

1. p is a wildcard variable

2. p and t are both a nullary symbol $a \in \Sigma$ where $p = a$, and $t = a$

3. p and t are both a n-ary symbol $a \in \Sigma$ where $p = a(r_1, ..., r_n)$, $t = a(g_1, ..., g_n)$, and for $1 \leq j \leq n$, r_j matches g_j

One might also notice that for p to match t, $p \geq t$, i.e. t is an instance of p. Furthermore, note that all expressions in a match set defined in definition 2.3.5 must match the same subject tree for it to be assigned to said subject tree. That is to say, for all tree patterns p in $match(t)$, where t is the subject tree, p matches t as defined in definition 2.3.6.

Patterns p, p' unify with one another (denoted $p \uparrow p'$) if there exists a subject tree that is an instance of both p, and p'. Other literature might refer to this as the independence relation.

2.3.2 Bottom-up (leaf to root)

Bottom-up, also called leaf-to-root, tree pattern matching refers to matching of pattern tree(s) to a subject tree by matching starting at the leaves (i.e. bottom), and going towards the root (i.e. going up). Hoffmann and O'Donnell [8] show few algorithms for bottom-up matching; however, the algorithms all share the same concept with minor optimizations. The key idea is to preprocess/precompile a lookup table for each possible symbol in Σ using the possible match sets as the state a node can be in. The table size for each symbol grows exponentially with respect to the arity of said symbol and number of match sets. For example, take n as number of possible match sets, then if $b \in \Sigma$ has an arity of 0, the resulting table would be merely a single value ($n^0 = 1$). If $a \in \Sigma$ has an arity of 2, then the resulting table would be 2 dimensional (n^2).

The table basically determines a node's state, or match set, using each of its child's match sets. The idea is that some match sets denote the scenario of a match for a pattern. In other words, if a node matches a match set that denotes a particular pattern, it can be concluded that the tree node matched said pattern.

To illustrate, take $\Sigma = \{a, b, c\}$, linear $x \in \mathcal{V}$ meaning all occurrences of x are indistinguishable, and patterns $p = a(a(x, x), b)$ and $p' = a(b, x)$. Then the possible match sets are:

1. $\{x\}$
2. $\{b, x\}$
3. $\{a(x, x), x\}$
4. $\{a(b, x), a(x, x), x\}$
5. $\{a(a(x, x), b), a(x, x), x\}$

Table 2.1: *Table for symbol a*

		right child				
		1	2	3	4	5
	1	3	3	3	3	3
	2	4	4	4	4	4
left child	3	3	5	3	3	3
	4	3	5	3	3	3
	5	3	5	3	3	3

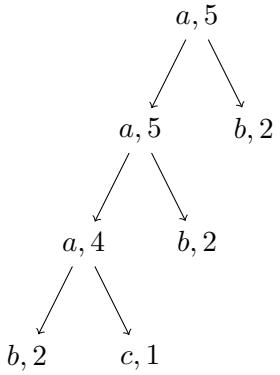


Figure 2.6: *Subject tree t*

The lookup table for symbol a is shown as table 2.1. Tables for symbols b and c are trivial as $match(b) = PF \cap \{b, x\}$ which is set 2, and $match(c) = PF \cap \{c, x\} = \{x\}$, thus set 1. Now to apply the aforementioned tables to an example subject tree, let subject tree $t = a(a(a(b, c), b), b)$. The result of going through the tables for each symbol in the subject tree t, a state is assigned and denoted after the tree's symbol in figure 2.6. As set 4 includes p', and set 5 includes p, it can be concluded that $t|_\Lambda$, and $t|_1$ match p, and that $t|_{1.1}$ matches p'.

Note about the example that variable wildcards of patterns are assumed to be linear within the pattern expression itself, as differences or similarities between variable wildcards can be post-computed easily by a comparison (if necessary). It is more convenient to reason about wildcards as can-be-anything variables in such a case.

Observe how the used tables compare to language parsing techniques such as LR(0) parsing [3]. In LR(0) parsing tables are also precompiled and used to determine a (context-free) language's validity and construction depending on language rules. Additionally, the tables in both LR(0) and bottom-up tree pattern matching relate to automata as state assignments using the lookup table can be compared to automata's state transitions.

Doing tree pattern matching using the table allows for a linear runtime complexity with respect to the size of the subject tree (i.e. $\Theta(|t|)$ for a subject tree t). Observe that the linear runtime complexity is due to the fact that each node in the subject tree has to be visited, hence the size of the subject tree.

The primary downside of this methodology is that it creates huge tables for each symbol which exponentially grow with respect to the arity of said symbol. Hoffmann and O'Donnell argue that restricting patterns to so called simple pattern forests meaning that the patterns matched don't unify with any other patterns will constrain exponential growth of states [8]. Remember that intuitively, match sets denote possible matching scenario's to be considered while matching. Then if $p, p' \in P$ where $p \neq p' \wedge p \uparrow p'$ would mean at least a match set with p, at least a match set with p', and at least a match set with both. This grows exponentially as more patterns unify with each other, therefore using simple pattern forests the exponential growth of table size will be avoided.

Table 2.2: *Right child mapping*

State	1	2	3	4	5
Mapping	1	2	1	1	1

Table 2.3: *Left child mapping*

State	1	2	3	4	5
Mapping	1	2	3	1	1

Table 2.4: *Improved table for symbol a*

		right child	
		1	2
left child	1	3	3
	2	4	4
	3	3	5

Despite that, it is not always feasible to constrain the patterns to only non-unifying patterns. Moreover, even the simple pattern forest tables can be optimized in size. Chase noticed that some entries on the axes of the table overlap in mapping [5]. For example, take table 2.1. The right child axis has overlap between $1, 3, 4, 5$, and the left child axis has overlap between $3, 4, 5$. These can be merged into a smaller equivalent table and mappings can be constructed for removed states. Doing so results in what's depicted in tables 2.3, 2.2, and 2.4.

Notice that the trade-off in Chase's improved table size is the extra lookup table jumps. Before, using the primary lookup table was constant, whereas now it's related to the arity of a symbol. In the aforementioned example where a has an arity of 2, 2 extra table lookups are necessary.

2.3.3 Top-down (root to leaf)

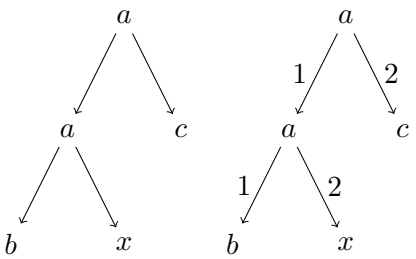


Figure 2.7: *Tree pattern and equivalent trie*

Similarly to how bottom-up tree pattern matching starts at the bottom (leaves) and matches trees whilst going upwards (to the root), top-down starts at the top (root) and matches trees whilst going downwards (to the leaves). Hoffmann and O'Donnell [8] also include ways to do top-down tree pattern matching. They reduce the tree patterns to strings and then solve the problem using string matching. First, tree patterns are converted to their equivalent trie structure where edges are labeled with the choice much like shown in figure 2.7. Then all paths in the trie from root to leaf are stringified and put into a set denoting the tree pattern. For the pattern $a(a(b, x), c)$ depicted in figure 2.7 where x is a wildcard variable, this set would look like $\{a1a1b, a1a2, a2c\}$. Note that the wildcard variable is omitted and that the number of resulting patterns is equal to the number of leaves. These strings are then used to construct an automaton using Aho and

Corasick's string matching algorithm [2]. The key idea is that all strings in a pattern's string set can successfully walk down the same root symbol. In such a case, walking means going from the root node downwards to a leaf node taking the path as denoted by the string.

Although Hoffmann and O'Donnell's algorithm works, reduction of the tree pattern matching to a string matching problem is unnecessary. An adaptive approach is proposed by Sekar, Ramesh, and Ramakrishnan [13]. In this approach, matching still occurs starting at the root, going to the leaves (i.e. top-down); however, the path going down a subject tree is heuristically determined at automaton build time such that a more efficient automaton can be constructed. This more efficient automaton can result in both reduction of matching time and reduction of state space. Additionally, this approach also supports priorities of patterns. Some patterns might have higher priority meaning that matches on lower priority patterns are delayed if any pattern of higher priority could still be a potential match.

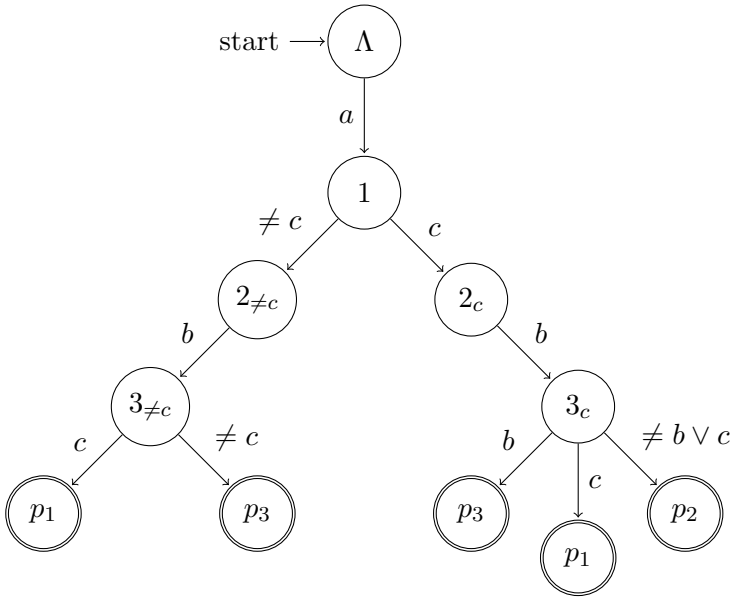


Figure 2.8: *Naive left-to-right DFA*

To illustrate, take patterns $p_1 = a(x, b, c), p_2 = a(c, b, b), p_3 = a(x, b, y)$ where p_i denotes a pattern with priority i, $\Sigma = \{a, b, c\}$, and $x, y \in \mathcal{V}$ are wildcard variables. Then figure 2.8 shows a naive left-to-right matching, states are labeled with the path being checked. Subscripts serve no purpose apart from identifying different states. Edges are the symbol being checked and $\neq z$ denotes all symbols that are not $z \in \Sigma \cup \mathcal{V}$. Moreover, accepting states are labeled with the pattern that is matched. Observe that left-to-right refers to the order in which the expressions are matched (starting from the left, going to the right).

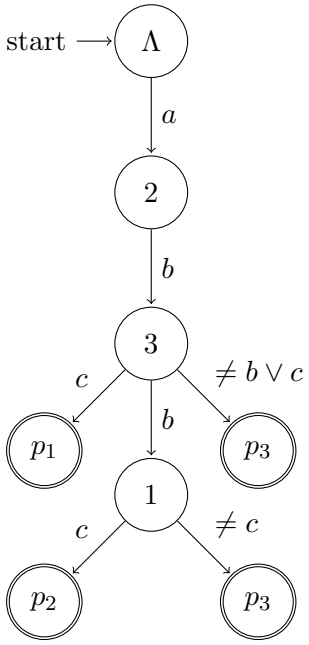


Figure 2.9: *Adaptive DFA*

Figure 2.9 shows the adaptive alternative of the same example. This figure shows an automaton where the matched subexpressions are out of order resulting in smaller state space size, and possibly faster matching which can finish within 3 checks (instead of the mandatory 4 in the figure 2.8). Deciding matching order of a set of patterns such that the algorithm constructs the most optimal automaton in size is NP-complete [13]. However, few greedy algorithms are discussed by Sekar, Ramesh, and Ramakrishnan such as using the path with the minimal amount of unique symbols in the matching patterns.

Figure 2.9 shows one of the problems with automata construction shown in [13]. Both edges containing $\neq$ result in the same match. The algorithm described could be augmented such that duplicated sub-automata can be avoided to some extent; however, an alternative methodology is described by Nedjah, Walter, and Eldridge [12]. This methodology introduces a notion of equivalence classes for states. It does so by keeping track of what patterns, and how much of said patterns have been matched thus far by each state in the automaton and takes a wildcard variable closure of this set for each state. The methodology is inspired by LALR parsing [3] which also constructs automata-like tables to compute the validity of a context-free grammar depending on rewriting rules. The result of each state's closure sets is that similar closure sets can be merged. The similarity condition uses the remaining non-checked symbols, and the expansion of wildcard variables to determine whether states are similar such that they can be merged.

The biggest drawback of [12] is that the constructed automaton does naive left-to-right matching which can be improved upon using an adaptive methodology as was shown in figures 2.8 and 2.9. An adaptive methodology using the same concept as [12] is described by Nedjah and Mourelle [11]. The naive left-to-right methodology in [12] kept track of what has been matched thus far using an auxiliary character $\bullet$ added in the pattern expressions (e.g. $a(a(b, \bullet c), y)$) where any symbol in $\Sigma \cup \mathcal{V}$ on the left of the auxiliary character has been matched, and all symbols on the right of the auxiliary character, has not been matched yet. The adaptive alternative described in [11]

introduces another auxiliary character ✓ to mark symbols that have been matched. For example, $a(a(b, \bullet c), y)$ of the naive left-to-right methodology has the adaptive equivalent $a^{\checkmark}(a^{\checkmark}(b^{\checkmark}, \bullet c), y)$ where both notations denote that pattern prefix $a(a(b$ has been matched so far. Of course, the idea of the adaptive variant is that these marks ✓ do not have to occur consecutively in the expression, nor are they constrained to only the left side of the tracking auxiliary character $\bullet$.

Chapter 3

Description

This chapter aims to motivate this project, and to describe the problem of this project. Section 3.1 motivates peephole optimizations and some of the fallbacks within LLVM's peephole optimizer. Section 3.2 describes the problem statement with its associated research question.

3.1 Motivation

The LLVM compiler suite supports, much like many other compiler suites, peephole optimizations. These peephole optimizations are small, static code transformations meaning that small snippets of code are analysed and, where possible, altered to gain runtime performance.

Not only is the peephole optimizer in LLVM used to gain performance, but many other optimization passes in LLVM depend upon the canonical form of code emitted by the peephole optimizer. Therefore, modifying the output of the peephole optimizer could negatively affect other optimizations, and thereby the performance of emitted code.

Despite the aforementioned responsibilities of the LLVM peephole optimizer, its runtime execution is less-than-stellar as it naively iterates through the code looking for a match on any of the peephole optimizations. If a match is found, it is then transformed into its equivalent, after which the peephole optimizer will iterate over the optimized code again to ensure no other peephole optimizations were made possible due to previous transformations. If any code has been altered in an iteration, the peephole optimizer will iterate again until no more modifications are applied to code during an iteration. This means that iterations run until a fixed-point is found where no more peephole optimizations can be applied.

Due to the naive implementation of the peephole optimizer for LLVM, compilation time is dominated by it. According to initial profiling done by engineers on llvm-dev[1] the peephole optimizer takes about 35% of total compilation time. The majority of the peephole optimization's execution seem to be caused by optimizing the code. Ideally, the peephole optimizer's runtime is decreased such that total compilation time is reduced. For this, both naive matching (and replacement), and fixed-point iterations of peephole optimizations ought to be changed such that fewer iterations are necessary and matching time is improved.

[1]LLVM mailing list, used by LLVM developers for questions on LLVM development and important LLVM design decisions. Link to related llvm-dev discussion on naive peephole optimizer: http://lists.llvm.org/pipermail/llvm-dev/2017-March/111257.html

Naive matching can be improved upon by using matching automata instead. The idea is that, because all peephole optimizations are known prior to compilation, the matching automata can be precompiled. Then, this matching automaton can be used during compiler execution to efficiently match peephole optimizations.

3.2 Problem statement

The goal of this project is to (1) automatically generate C++ code for LLVM's peephole optimizer using Alive, and (2) speed up execution time of the peephole optimizer in LLVM.

The purpose of (1) is to help developers who intend to add new peephole optimizations to LLVM. Currently, adding new peephole optimizations require the developer to:

1. Show correctness of newly developed peephole optimization(s) e.g. using Alive
2. Have an understanding of LLVM IR pattern matching and LLVM IR code transformations internals
3. Have an understanding of LLVM's InstCombine and InstSimplify passes (with all of its caveats)

The ideal workflow for developers who intend to add new peephole optimizations to LLVM is to define the peephole optimization's transformation, supply said definition of a peephole optimization's transformation to a tool which will then automatically prove the correctness of said transformation, and generate C++ code of the peephole optimization's transformation which is ready for compilation when included in LLVM. This workflow omits the need to know about LLVM internals and passes in detail.

The purpose of (2) is to accelerate compilation time of LLVM which uses the generated C++ code in (1). Since C++ code generated at (1) does not require the developer of new peephole optimizations to have deep knowledge on LLVM internals it is possible to restructure the generated C++ code to a more efficient, albeit possibly less clear, alternative.

A definition of a peephole optimization requires the following information for both verification and code generation:

1. Precondition: requirements on the variables found within the peephole optimization that have to hold (can be omitted, in which case the precondition can be considered 'true')
2. Source code pattern: snippet of code that has to be matched for the peephole optimization
3. Target code snippet: if both the precondition holds and a match on the source code pattern has been found, then the code will be transformed to the target code snippet

The peephole optimization matching problem can be described as the problem of matching and transforming code. Given the aforementioned necessary peephole optimization definition information, 1 has to hold, a match has to be found for 2, and transformation code has to be invoked for 3 when both 1 holds and 2 has been matched. As the current peephole optimizer in LLVM shows the potential to be optimized in terms

of execution speed by use of tree pattern matching, the tree pattern matching problem needs to be extended and constrained for the peephole optimization problem.

The resulting question is: how can tree pattern matching methodologies be extended to solve the peephole optimization matching problem.

In the end, this thesis aims to describe how extensions can be applied to tree pattern matching to conform to additional constraints (in this case, the constraints introduced by peephole optimizations). This way, developers can implement similar tree pattern matching methodologies which introduce constraints similar to the ones introduced by peephole optimizations.

Chapter 4

Tree pattern matching in Alive

Shown in figure 2.3 is the interaction between Alive and LLVM in terms of compilation that can be considered a compiler-compiler. This phrase refers to the fact that code has to be generated by Alive, for the LLVM compiler toolchain. In such a sense, Alive is a compiler for a compiler. Though Alive is not yet at the point where the shown workflow is extensively used, it is the ideal workflow to allow LLVM developers a more structured way of extending peephole optimizations in LLVM.

This chapter will explain the necessary extension to tree pattern matching such that tree pattern matching can be used for peephole optimization matching. Furthermore, it will explain some implementation specific differences between top-down and bottom-up approaches used for peephole optimization(s) in Alive.

4.1 Additional constraints for tree pattern matching

Tree pattern matching can be extended to support peephole optimization matching and transforming. This section aims to explore some additional constraints that come with peephole optimization matching compared to tree pattern matching.

4.1.1 Linearity of wildcards

Most literature on tree pattern matching, such as [8, 13, 5], puts barely any constraints on pattern trees. For example, it is assumed wildcards that occur in tree patterns are linear, meaning they are indistinguishable from one another (i.e. occurrences of wildcards might not be comparable). Moreover, only a single type of wildcard is considered. These choices are sensible within the context of tree pattern matching. For example, if linearity was not assumed, then pattern matching can be considered Directed Acyclic Graph (DAG) pattern matching instead of tree pattern matching.

Figure 4.1: *Non-linear tree l*

Shown in figure 4.1 is a non-linear tree l where add is a symbol with an arity of 2 and a is a variable. Note that variable a is a constrained wildcard and is distinguishable from

other variables. Because of this property, it is now constrained such that both children of $l|_\Lambda$ must reference the same value. Tree pattern matching methodologies described in [8] do not consider non-linear trees, therefore tree pattern matching needs to be extended to take into account such constraints.

```
%r = sub %x, C
     =>
%r = add %x, -C
```

Listing 4.1: *Alive DSL example with constant wildcard*

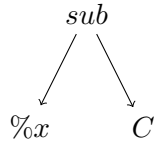


Figure 4.2: *Tree with constant wildcard*

4.1.2 Constant wildcards

Additionally, apart from variable wildcards there are also constant wildcards which can be considered wildcards that are even more constrained. These are the wildcards that refer to only constant values. For example, take the peephole optimization in listing 4.1, the $\%x$ is a variable wildcard and C is a constant wildcard. To illustrate, take the following expressions that match the source tree pattern as defined in listing 4.1:

- $sub(add(x, y), 5)$, where $x, y \in \mathcal{V}$
- $sub(x, 8)$, where $x \in \mathcal{V}$
- $sub(92, 633)$

And the following expressions that do not match the source tree pattern as defined in listing 4.1:

- $sub(x, y)$, where $x, y \in \mathcal{V}$
- $sub(5, x)$, where $x \in \mathcal{V}$
- $sub(x, add(y, z))$, where $x, y, z \in \mathcal{V}$

As can be observed from the example expressions, constant wildcards, which are always prefixed with a C, are constrained to constant values whereas regular variable wildcards are not. Note that also constant wildcards are non-linear similarly to variable wildcards meaning that using the same name multiple times in a pattern results in explicitly matching for the same constant value. For example pattern $add(C, C)$ will only match expressions with an *add* function and two constant values that are the same. To differentiate between multiple constant wildcards, a differing suffix in form of an integer is applied to the constant wildcards. For example, pattern $add(C1, C2)$ matches expressions with an *add* function and can have two different constant values.

This additional constrained constant wildcard alters the concept of expressions as introduced in definition 2.3.1. Now, an expression is a recursively constructed formula of variable symbols $\mathcal{V}$, constant wildcard symbols $\mathcal{C}$, and function symbols from an alphabet Σ with each symbol a finite arity (i.e. number of arguments) where an arity of 0 dictates a constant literal value. More formally:

Definition 4.1.1. An expression is either:

1. a variable from the set of variables $\mathcal{V}$, or

2. a constant wildcard from the set of constant values $\mathcal{C}$, or

3. a nullary expression, i.e. a constant literal, or

4. an n-ary expression where $n > 0$ and each of the n arguments is an expression

The change in expressions described in definition 4.1.1 also affects other important aspects of tree pattern matching. For example, an alternative concept of substitution must now be added to allow substitutions of constant wildcards and constant values. Now, substitutions are mappings from variable wildcards to expressions, and constant wildcards to constant values. Take $\sigma = \{x \to sub(a, b), C1 \to 8\}$, where $\Sigma = \{add, sub\}$, $a, b, x \in \mathcal{V}$, and $C1 \in \mathcal{C}$, then $\sigma(add(x, C1)) = add(sub(a, b), 8))$.

The concept of subsumption in definition 2.3.3 does not change. However, the addition of constant wildcards means that every variable wildcard subsumes every constant wildcard, every constant wildcard subsumes every constant value, and by transitivity, every variable wildcard subsumes every constant value (i.e. $x \geq C1 \geq 0$, where $x \in \mathcal{V}$ and $C1 \in \mathcal{C}$).

4.1.3 Preconditions

Tree pattern matching focusses on matching a pattern tree to a subject tree, where the pattern tree can abstract some subtrees that might occur in the subject tree. These abstractions were introduced in the previous sections in the shape of constrained wildcards. However, these constrained wildcards might need some further constraining within the context of a pattern tree. For such pattern trees, the addition of a precondition is necessary to explicitly constrain the wildcards used within the tree(s).

```
Pre: isPowerOf2(C1)
%r = mul %x, C1
   =>
%r = shl %x, log2(C1)
```

Listing 4.2: *Alive DSL precondition example*

Shown in listing 4.2 is a peephole optimization in the Alive DSL which contains a precondition. This precondition also uses the built-in Alive function 'isPowerOf2' to constrain the values of $C1$ to only constant values that are a power of 2 (i.e. 1, 2, 4, 8, 16, ...). For this peephole optimization, this precondition is necessary for its correctness. It is easy to show a counterexample if this peephole optimization would not contain this precondition.

Given the following expression: $mul(2, 5) = 10$, then a transformation according to the peephole optimization defined in listing 4.2 (ignoring the precondition) would result in $shl(2, log2(5)) \approx shl(2, 2.321)$. Of course, shifting a value by a non-integer constant value is not possible.

Tree pattern matching methodologies will have to be extended to support these preconditions considering that merely tree pattern matching is not sufficient for peephole optimization transformations. Furthermore, preconditions exist in two different forms: implicit preconditions which are embedded in the peephole optimization source pattern, and explicit preconditions which are defined in the precondition field of a peephole optimization definition. For example, take multiple occurrences of variables or constant wildcards that need to be checked for equality, for a transformation to be correct.

4.1.4 Transformation

For peephole optimization transformations, the matching phase is only the first half of the peephole optimization. After matching, the code transformation itself needs to take place. Generally, transformations require subexpressions of the peephole optimization's source pattern (i.e. the expression matched) to construct the peephole optimization's target expression (i.e. the transformed expression).

The easiest way to accomplish this is to construct a new tree according to the peephole optimization's target expression and to replace the subject tree matched in its entirety. In some cases constructing the target code tree might not even be necessary. For example, when the target code expression is an already existing sub-tree.

4.2 Tree pattern ordering

Tree patterns can be relatable, as shown using the tree subsumption relation described in definition 2.3.3. Moreover, tree patterns can use relations such as these to derive a notion of order between tree patterns. This section aims to explain such orderings and priorities between tree patterns.

4.2.1 Partial ordering

Definition 2.3.3 describes the subsumption relation between trees. Note that this binary relation is not applicable to every pair of trees. For example, take $t_1 = add(x, y)$, and $t_2 = sub(a, b)$ where $a, b, x, y \in \mathcal{V}$ and $\Sigma = \{add, sub\}$, then $t_1 \not\geq t_2 \wedge t_1 \not\leq t_2$. Because not every tree pair is comparable, this type of ordering is also called a partial ordering [10]. This means that the subsumption relation's characteristics are, in accordance with partial ordering, as follows:

The subsumption binary relation is:

1. reflexive: for any tree t, $t \geq t$ i.e. t subsumes itself
2. anti-symmetric: for all trees t_1 and t_2, if $t_1 \geq t_2 \wedge t_2 \geq t_1$ then $t_1 = t_2$ i.e. two distinct trees cannot subsume one another
3. transitive: for all trees t_1, t_2, and t_3, if $t_1 \geq t_2 \wedge t_2 \geq t_3$ then $t_1 \geq t_3$ i.e. a tree subsumes another tree if a third tree is subsumed by one but subsumes the other

4.2.2 Priority

```
Pre: (computeKnownZeroBits(%x) | ↩
    computeKnownZeroBits(%y)) == -1
%r = add %x, %y
    =>
%r = or %x, %y
```

Listing 4.3: *Alive DSL general source pattern tree*

```
%nx = xor %x, -1
%r = add %nx, C
    =>
%r = sub C-1, %x
```

Listing 4.4: *Alive DSL less general source pattern tree*

For peephole optimizations, the most specific (i.e. least subsumed) source tree pattern is given priority over more general source tree patterns. For example, the source pattern of the peephole optimization shown in listing 4.3 is more general than the source pattern shown in listing 4.4 (i.e. $add(x, y) \geq add(xor(x, -1), C)$). When considering

only the tree patterns of both aforementioned listings, without transformation or preconditions, it can be said that when a match is found for listing 4.4, listing 4.3 will always match, too. In that sense, listing 4.3 is a more general version of listing 4.4.

Because of this subsumption relation between the source patterns of listings 4.3 and 4.4 a priority needs to be given in case both match at a subject tree. If listing 4.3 would always emit more efficient code compared to listing 4.4 it would be better to omit the peephole optimization of listing 4.4 entirely. It is therefore assumed that less specific peephole optimizations generate more efficient code and will therefore be prioritized over the more general peephole optimization pattern trees. However, it would be ideal to still use the peephole optimizations with the more subsuming source pattern as fallback in case preconditions for the less general do not hold.

The ordering of subsuming source patterns can be visualized using an immediate subsumption graph as described in [8]. This graph is a DAG where each vertex is a pattern tree, and each directed edge points to another vertex containing the immediate subsuming pattern tree of the source vertex's pattern tree. An immediate subsuming pattern tree is a pattern tree that subsumes another pattern tree without other pattern trees that transitively occur between said trees. In other words, take pattern trees t_1, t_2, t_3, then t_1 immediately subsumes t_2 if there exist no t_3 for which $t_1 > t_3 \wedge t_3 > t_2$, and it holds that $t_1 > t_2$. The immediate subsumption graph therefore depicts the transitive reduction graph.

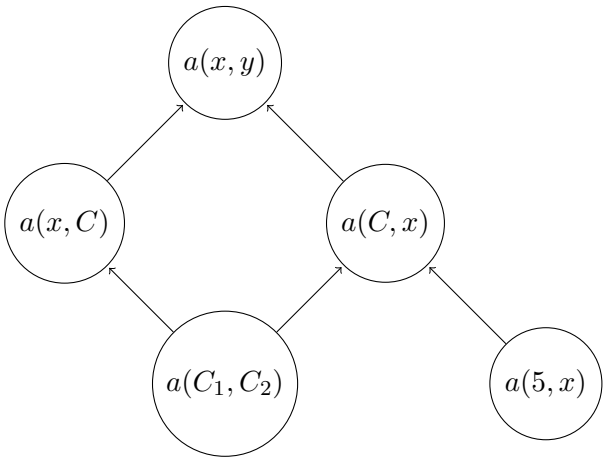


Figure 4.3: *Immediate subsumption graph example*

Figure 4.3 shows an immediate subsumption graph for expressions $a(x, y)$, $a(x, C)$, $a(C, x)$, $a(C_1, C_2)$, $a(5, x)$ where $a \in \Sigma$, $x, y \in \mathcal{V}$ and $C, C_1, C_2 \in \mathcal{C}$. Note that no direct edge between $a(x, y)$ and $a(5, x)$ even tough $a(x, y) > a(5, x)$. This is because there exists a tree, $a(C, x)$, for which $a(x, y) > a(C, x)$ and $a(C, x) > a(5, x)$, therefore the edge between $a(x, y)$ and $a(5, x)$ is omitted (i.e. omitted because it is not immediately subsuming). Furthermore, figure 4.3 shows that some trees are incomparable but still relatable to one another. Take pattern trees $a(C, x)$ and $a(x, C)$, these trees do not compare as neither subsumes the other; however, there exists more specific trees that are subsumed by both trees which is in accordance to the unification described in section 2.3.1.

4.2.3 Topological sort

The immediate subsumption graph used for pattern tree priority still contains patterns that are not comparable, i.e. unifying patterns. These unifying patterns are still assumed to have a distinct priority between them. To achieve this, the immedate subsumption graph is linearized using topological sort. Topological sort ensures the linear order using a DAG by taking vertices (pattern trees in our case) that do not have any other vertex refer to it and puts them in a sequence. This sequence will contain the vertices where the order between vertices does not break the subsumption relation. That is to say, a tree pattern p in the sequence can only be of higher priority if for all tree patterns q of lower priority q subsumes p, or p and q unify. When a vertex is put in the sequence it is then removed from the DAG together with its outgoing edges to allow other vertices to be appended to the sequence. The order of the vertices available to append to the sequence is arbitrary.

To illustrate, take the immediate subsumption graph depicted in figure 4.3. Initially, the only vertices available to be appended to the sequence are vertices $a(5, x)$ and $a(C_1, C_2)$. Putting these vertices in the sequence (arbitrarily) and removing them from the immediate subsumption graph will result in enabling the following vertices to be added: $a(x, C)$ and $a(C, x)$. Removing these vertices from the immediate subsumption graph will result in one remaining vertex, $a(x, y)$. Then, the sequence created may look as follows:

1. $a(5, x)$
2. $a(C_1, C_2)$
3. $a(x, C)$
4. $a(C, x)$
5. $a(x, y)$

Where a lower number denotes a higher priority. Note that the order of the available vertices at each iteration is allowed to be arbitrary as any permutation between these do not break the subsumption relation order. In other words, a pattern tree within this sequence will always occur before any pattern tree that subsumes it. For example, if $a(5, x)$ and $a(C_1, C_2)$ swap their positions within the sequence, the subsumption relation is not broken and they are, therefore, allowed to be ordered arbitrarily.

4.3 Top-down

Previously, Alive was to be considered a compiler-compiler. In case of top-down peephole optimization matching, this goes even further as the compiler analogy fits figure 2.2 which shows an abstract design of a compiler with its front-end, optimizer, and back-end. Alive's top-down peephole optimization matching shared the same front-end as used for formal verification of peephole optimizations. Using the parsed peephole optimization definitions, an automaton is constructed. This automaton can be considered the IR of top-down peephole optimization matching. It is, to some extent, possible to iterate over the generated automaton for further optimizations given that none of the functionality the automaton describes is lost. Then, the back-end takes in an automaton and generates its equivalent in C++, for use within LLVM.

This section aims to describe the design decisions for an initial implementation of top-down peephole optimization matching and transforming in Alive.

4.3.1 Methodology

For top-down peephole optimization matching, the methodology described by Sekar, Ramesh, and Ramakrishnan [13] was extended to take into account the constraints introduced in section 4.1. The algorithm as introduced in [13] builds the automaton recursively, state by state. The augmented algorithm constructs the automaton in a similar manner, recursively and one state at a time. This section aims to introduce some augmentations to the original algorithm for peephole optimization matching and transforming.

Final state transformations

One of the augmentations of the algorithm is the labeling of the final states. Originally, a set of possibly matching tree patterns is assigned to a final state. However, it is only possible to apply a single transformation with peephole optimizations. In other words, if the set of possibly matching trees contains multiple trees which unify with one another a choice has to be made between the unifying trees on which transformation will be applied. To make a choice, the tree pattern with the highest associated priority is used as match. Remember that final states are required to contain context on which tree pattern has been matched which is why labeling the final state is necessary.

Duplicate sub-automata reduction

Using the original algorithm, some duplicate sub-automata might be constructed within an automaton. These duplicate sub-automata result in a bigger automaton with more states. Preferably, unnecessary duplicate sub-automata are reduced such that the number of states are suppressed.

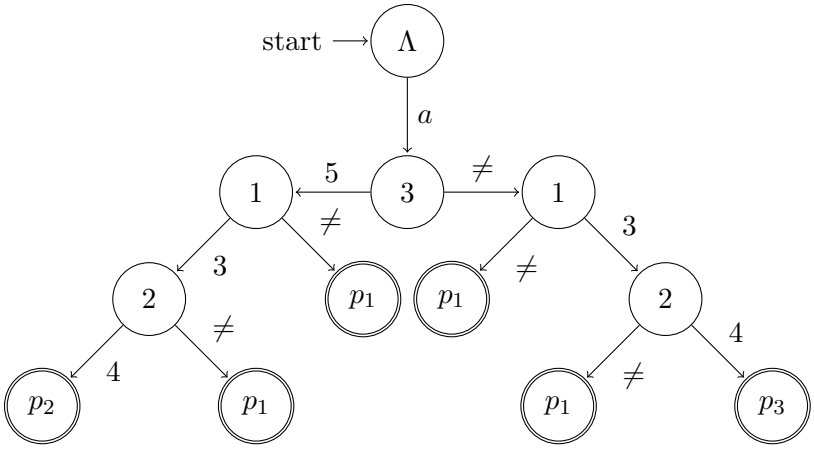


Figure 4.4: *DFA constructed with duplicate sub-automata*

For example, take $\Sigma = \{a, 3, 4, 5\}$, $x, y, z \in \mathcal{V}$, and tree patterns $p_1 = a(x, y, z)$, $p_2 = a(3, 4, 5)$, $p_3 = a(3, 4, z)$, then the constructed top-down matching automaton is shown in figure 4.4. This automaton shows the duplicate sub-automata for final states p_1. A small example such as the one depicted in figure 4.4 contains only a few overlapping sub-automata with the sub-automata size consisting of only a single state. However, with the number of tree patterns increasing together with the size of each tree pattern it would be preferred to merge these duplicates somehow.

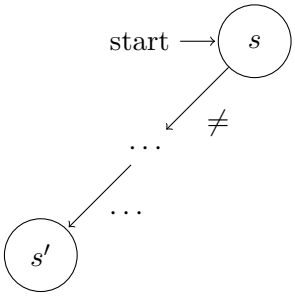


Figure 4.5: *Example without diverging* $\neq$

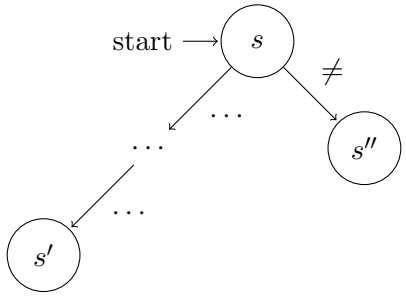


Figure 4.6: *Example with diverging* $\neq$

The extension that avoids unnecessary duplicate sub-automata uses the closest ancestor node with a diverging $\neq$ transition. This ancestor can be found by backtracking starting from the currently generated state towards the initial state. During backtracking, the first state found which transitions to a different (i.e. diverging) state not on the backtracked path is an ancestor with a diverging $\neq$. Note that it is possible to have no diverging $\neq$ on the path from a particular state towards its initial state. Figure 4.5 shows a situation where no diverging $\neq$ exists in the path from s' towards its initial state s. A $\neq$ transition exists on the path itself but none exist that diverge and therefore, there is no ancestor node with a diverging $\neq$ transition. Figure 4.6, however, shows a situation where an ancestor has a diverging $\neq$ as the path from s' to initial state s has a diverging $\neq$ from s to s''. As multiple of these diverging $\neq$ transitions might exist on a path towards the initial state, the one of interest is the first diverging $\neq$ found on the path from a state to the initial state. This closest ancestor with a diverging $\neq$ can be used by taking the state at the destination of this $\neq$ transition. This state can then be used as state to transition to on $\neq$ transitions for the particular state being processed.

Searching for the destination state of the diverging $\neq$ is also referred to as finding the closest diverging sink state as the $\neq$ transition is considered to be a sink transition. The $\neq$ transition encompasses all remaining transitions on input in Σ not yet explicitly covered in that state, hence sink transition.

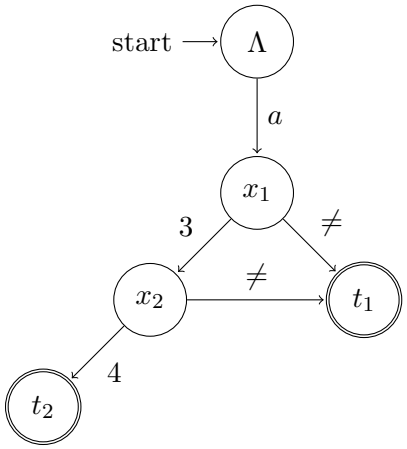


Figure 4.7: *DFA constructed using ClosestDivergingSink*

To illustrate, take $\Sigma = \{a, 3, 4, 5\}$, $x, y \in \mathcal{V}$, and tree patterns $t_1 = a(x, y)$, $t_2 = a(3, 4)$, then figure 4.7 shows the resulting automaton. Note that during construction of state x_2, the closest diverging sink state is t_1 since on the path from state x_2 to its initial

state Λ, the state x_1 contains a diverging $\neq$ transition towards t_1. Then an additional $\neq$ transition is added from x_2 to t_1 as fallback for x_2.

An input expression example for the automaton in figure 4.7 which takes this fallback option for x_2 is subject expression $t_s = a(3, 5)$. When using t_s as input the path taken in the automaton will be $\Lambda \to x_1 \to x_2 \to t_1$. The transition between x_1 and x_2 is taken since the symbol at $t_s|_1$ is 3 which has an explicit transition to x_2. However, the symbol at $t_s|_2$ does not have an explicit transition, hence the sink transition $\neq$ is taken. This transition then finds a tree pattern match for t_s where t_s matches tree pattern t_1.

Algorithm 2 ClosestDivergingSink function

input:

$dfa : (Q, \Sigma, \delta, q_0, F)$	-	deterministic finite automaton searched for its closest diverging sink state
$s : Q$	-	state whose closest diverging sink state is to be located

vars:

$n : \mathbb{N}$	-	the number of steps for the transitive closure of δ^n
$\delta^{n \in \mathbb{N}} : Q \times \Sigma \to Q$	-	Transitive closure of transitions in δ, up until n steps

$$CDS(dfa, s) = \begin{cases} q, & \text{if } \exists s, s', s'' \in Q.\ \exists a \in \Sigma \cup \{\neq\}.\ (s', a) \to s \in \delta^n \wedge (s', \neq) \to \\ & s'' \in \delta, \text{ where } q \text{ is state } s'' \text{ with the maximum associated } n \\ \text{nil}, & \text{otherwise} \end{cases}$$

Algorithm 2 shows the function for finding the closest diverging sink state (CDS). This function takes in a state whose closest diverging sink state is being searched and a deterministic finite automaton of which its quintuple $(Q, \Sigma, \delta, q_0, F)$ is used for searching. This function returns q if a diverging sink state has been found, and nil otherwise. The return value q denotes the state reached using a $\neq$ sink transition from its parent state which is located on the path between the initial state and input state s and has the highest number of transitional steps between the initial state to the parent state (which, in turn, results in the lowest number of steps from input state s to the parent state).

Variable and constant wildcards

The original algorithm assumes only a single kind of unconstrained wildcard type. However, as discussed in section 4.1, because of peephole optimizations, this concept of wildcards needs to be extended and further constrained with variable wildcards and constant wildcards. For automaton construction, every sink transition will now be expanded to diverge between a path for variable wildcards and a path for constant wildcards.

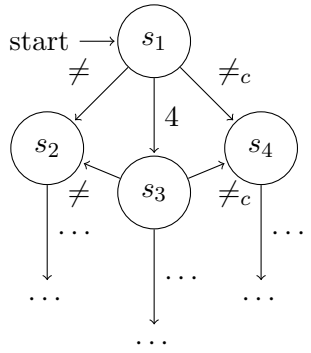


Figure 4.8: *Multiple wildcards with ambiguous ClosestDivergingSink*

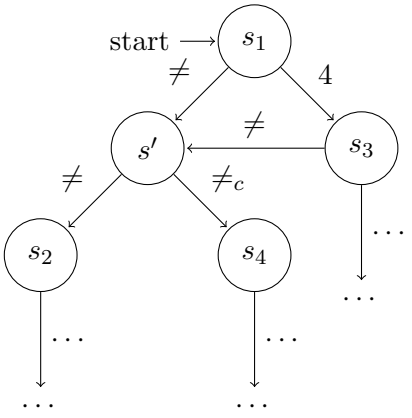


Figure 4.9: *Multiple wildcards with unambiguous ClosestDivergingSink*

To accommodate wildcards, the original algorithm uses sink (i.e. $\neq$) transitions. The obvious solution would be to extend this concept and introduce a constant wildcard sink $\neq_c$ that transitions in case of constant literals that do not have an explicit transition from a particular state; however, this might result in erroneous functionality when processing the closest diverging sink state. To illustrate, take the automaton shown in figure 4.8 which naively introduces a $\neq_c$ transition similar to how a $\neq$ transition is introduced. In this case, if state s_3 needs a fallback option using the closest diverging sink state, only state s_2 would be used as fallback; however, state s_4 should also be used as fallback state in case the transition is on a constant literal that has no explicit transition in state s_3. In other words, an additional closest diverging sink for constant wildcards would need to be introduced for this methodology.

Instead, an alternative is used which does not require searching for the closest diverging constant wildcard sink state. This alternative constructs an additional state which is used only for diverging variable wildcards and constant wildcards. This way the closest diverging sink state suffices as it results in the state which then splits off between the variable wildcard or constant wildcard. To illustrate, the automaton in figure 4.9 depicts the additional state as state s'. This state s' is the closest diverging sink state for state s_3. Without the need for state s_3 to search for its closest diverging constant wildcard sink state, the state can still fallback on either variable wildcards and constant wildcards. Note that states s_1 and s' will be labeled with the same coordinate. Recall that this coordinate depicts the coordinate within the subject tree that is to be checked in said state.

Algorithm 3 CreateSink pseudocode

input:

$dfa : (Q, \Sigma, \delta, q_0, F)$	-	deterministic finite automaton being constructed
$s : state$	-	when in unlabeled state s, the prefix expression e will have been processed
$e : \mathcal{E}$	-	prefix expression, used keep track of what's processed so far
$V : \{\mathcal{E}\}$	-	set of patterns that could match prefix expression e
$co : path$	-	string of edge choices

vars:

$root : \mathcal{E} \to \mathcal{S}$	-	gives the symbol at the root of a expression
$sink : Q$	-	the closest diverging sink state relative to s

1: **function** CREATESINK(dfa, s, e, V, co)
2: label s with co
3: **if** $\exists v \in V.\ root(v|_{co}) \in \mathcal{V}$ **then**
4: Create fresh state $s_{\neq}$
5: $\delta := \delta \cup \{(s, \neq) \to s_{\neq}\}$ ▷ $\neq$ denotes any symbol not in $\mathcal{C}$
6: BUILD($s_{\neq}, e_{co}[\neq], \{p \in V \mid root(p|_{co}) \in \mathcal{V}\}$)
▷ the $\neq$ sub-expression denotes a variable occurrence
7: **else**
8: $sink :=$ CDS(dfa, s)
9: **if** $sink \neq nil$ **then**
10: $\delta := \delta \cup \{(s, \neq) \to sink\}$ ▷ $\neq$ denotes any symbol not a literal constant
11: **if** $\exists v \in V.\ root(v|_{co}) \in \mathcal{C}$ **then**
12: Create fresh state $s_{\neq_c}$
13: $\delta := \delta \cup \{(s, \neq_c) \to s_{\neq_c}\}$ ▷ $\neq_c$ denotes any symbol in $\mathcal{C}$
14: BUILD($s_{\neq_c}, e_{co}[\neq_c], \{p \in V \mid root(p|_{co}) \in \mathcal{C}\}$)
▷ the $\neq_c$ sub-expression denotes a constant wildcard occurrence

Shown in algorithm 3 is the CreateSink function pseudocode. The input for this function are:

1. dfa, the DFA currently being constructed.
2. s, the unlabeled state used to transition to new fresh states depending on the tree patterns.
3. e, the (prefix) tree pattern which has the shape of the tree matched thus far, i.e. when in state s the tree matched thus far is e.
4. V, tree patterns whose symbol at coordinate co is either a variable wildcard in $\mathcal{V}$ or a constant wildcard in $\mathcal{C}$.
5. co, path currently being built upon in (prefix) tree pattern e.

No output is necessary for this function as construction happens locally using the input DFA.

Since this function takes care of diverging between variable wildcards and constant wildcards, it is assumed that all tree patterns in input V have either a symbol in $\mathcal{V}$ or a symbol in $\mathcal{C}$ at coordinate co. That is to say that the following is assumed to hold:

$$\forall v \in V.\ root(v|_{co}) \in \mathcal{V} \vee root(v|_{co}) \in \mathcal{C}$$

The procedure described in algorithm 3 shows that the unlabeled state s is first labeled using the input path *co*. This will be the same label as its parent state, of which it only has one at this point during construction, as can be observed in line 2. Then, a check is done to see whether a new sink state is necessary. This is done by checking whether there exists a variable wildcard at coordinate *co* for any of the patterns in V. If this is the case, a fresh state is created, a $\neq$ (i.e. sink) transition is added from s to the fresh state, and the fresh state is then recursed on using the recursive automaton construction function *Build*. If no variable wildcard exists at coordinate *co* for any of the pattern in V, then the closest diverging sink state is searched for using the CDS function. If such a state exists, it will be used to $\neq$ transition towards from s; however, if none exists no additional transitions will be added using the regular variable wildcard transition $\neq$. Then, a similar process is used for constant wildcards starting from line 11 in the algorithm. Here, a check is made to see whether there exists a constant wildcard at coordinate *co* for any of the patterns in V. If this is the case, a fresh state is created, a transition is made from s to the fresh state using a $\neq_c$ transition (which is a transition only taken when the subject tree used as automaton input contains a constant literal at coordinate *co*), and the fresh state is then recursed on using the recursive automaton construction function *Build*; however, if no constant wildcard exists at coordinate *co* for any of the patterns in V no modifications are applied to the input DFA. Note that it is important that the $\neq$ construction has priority over $\neq_c$ construction as the path down the $\neq_c$ transition might use the $\neq$ transition in state s for its closest diverging sink state.

Note that not every invocation of *CreateSink* will result in two additional fresh states. Creation of these states depend upon the availability of tree patterns in V with either a variable wildcard or constant wildcard at coordinate *co*. This means the following sub-automata are possible by *CreateSink*'s construction:

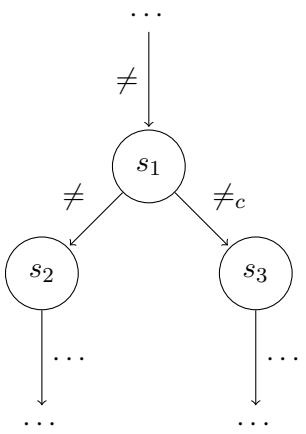


Figure 4.10: *Both $\neq$ and $\neq_c$ transitions from sink state s_1*

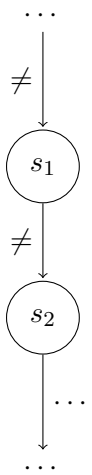


Figure 4.11: *Only a $\neq$ transition from sink state s_1*

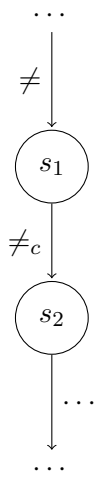


Figure 4.12: *Only a $\neq_c$ transition from sink state s_1*

The automaton in figure 4.10 shows the case where both variable wildcards and constant wildcards occur for patterns in V, at coordinate *co*. Note that the same closest diverging sink state search can still be applied. For instance, the closest diverging sink state of state s_3 is state s_2 since its parent is on the path from s_3 to the initial state, and the transition from the parent to s_2 is a $\neq$ transition.

The automaton in figure 4.11 shows the case where only variable wildcards occur for all patterns in V, at coordinate *co*. Note that now, a replica exists for state s_1 as s_1's parent has the same path string label as s_1. In other words, the same path coordinate is checked twice with the same $\neq$ sink transition. These duplicated transitions can be removed using a subsequent optimization pass through the automaton which detects this inefficiency and promptly removes it.

The automaton in figure 4.12 shows the final possible scenario where only constant wildcards occur for all patterns in V, at coordinate *co*. Here, the inefficiency of the unnecessary $\neq$ transition towards state s_1 could be removed after construction of the automaton. However, during construction this $\neq$ might be used for another state's closest diverging sink state search.

Top-down construction function

So far, all discussed automaton related construction algorithms are extensions of the existing original construction procedure; however, this procedure is still required to be modified to support these extensions.

Algorithm 4 Top-down Build pseudocode

input:

$dfa : (Q, \Sigma, \delta, q_0, F)$	-	deterministic finite automaton being constructed
$s : Q$	-	when in unlabeled state s, the prefix expression e will have been processed
$e : \mathcal{E}$	-	prefix expression, used keep track of expression tree processed so far
$P : \{\mathcal{E}\}$	-	set of patterns that could match prefix expression e

vars:

$M : \{\mathcal{E}\}$	-	patterns of which e is an instance
$c : path$	-	string of edge choices
$FS : \{\mathcal{S}\}$	-	symbols of all patterns in P at coordinate c
$max_{prio} : \{\mathcal{E}\} \to \mathcal{E}$	-	gives the expression with the highest priority
$prio : \mathcal{E} \to \mathbb{N}$	-	gives the priority of a certain expression
$nextop : \mathcal{E} \to path$	-	gives the next operand to match as a path
$root : \mathcal{E} \to \mathcal{S}$	-	gives the symbol at the root of a expression
$ar : \mathcal{E} \to \mathbb{N}$	-	gives arity (i.e. rank) of an expression
$sink : Q$	-	the closest diverging sink state relative to s

1: **function** BUILD(dfa, s, e, P)
2: $M := \{p \in P \mid e \leq p\}$
3: **if** $M \neq \emptyset$ and $\forall p \in P.\ \exists p' \in M.\ prio(p') \geq prio(p) \wedge p' \not> p$ **then**
4: label s as final state for expression $max_{prio}(M)$
5: $F := F \cup \{s\}$
6: **else**
7: $c := nextop(e, P)$
8: Label s with c
9: $FS := \{f \mid \exists p \in P.\ root(p|_c) = f \wedge \neg(f \in \mathcal{V} \wedge f \in \mathcal{C})\}$
10: **if** $\exists p \in P.\ root(p|_c) \in \mathcal{V} \vee root(p|_c) \in \mathcal{C}$ **then**
11: Create fresh state $s_{\neg FS}$
12: $\delta := \delta \cup \{(s, \neq) \to s_{\neg FS}\}$ ▷ $\neq$ denotes any symbol not in FS
13: CREATESINK($s_{\neg FS}, e_c[\neq], \{p \in P \mid root(p|_c) \in \mathcal{V} \vee root(p|_c) \in \mathcal{C}\}, c$)
▷ the $\neq$ sub-expression denotes a variable occurrence
14: **else**
15: $sink :=$ CDS(dfa, s)
16: **if** $sink \neq nil$ **then**
17: $\delta := \delta \cup \{(s, \neq) \to sink\}$ ▷ $\neq$ denotes any symbol not in FS
18: **for all** $f \in FS$ **do**
19: Create fresh state s_f
20: $\delta := \delta \cup \{(s, f) \to s_f\}$
21: BUILD($s_f, e_c[f(x_1, ...x_{ar(f)})], \{p \in P \mid root(p|_c) = f\}$)

Shown in algorithm 4 is the pseudocode for the augmented automaton construction function *Build*. The input of this function is similar to the input for the *CreateSink* function in algorithm 3:

1. dfa, the DFA currently being constructed.

2. s, the unlabeled state used to either mark as final state or used to transition to fresh states depending on the tree patterns in P.

3. e, the (prefix) tree pattern which has the shape of the tree matched thus far, i.e. when in state s the tree matched thus far is e.

4. P, the tree patterns that match e, i.e. every tree patterns in P is subsumed by e.

Similarly to the $CreateSink$ function, no output is necessary as the construction of the DFA happens locally by augmenting dfa.

Lines 2, 3, 4, and 5 in algorithm 4 show the process of marking state s as a final state. On line 2 the set M is constructed containing all patterns in P which subsume (prefix) tree pattern e. Then, line 3 has the acceptance condition which has to hold for state s to be marked as a final state. This acceptance condition holds when M is not empty, the tree patterns in M are all of higher or equal priority than any tree pattern in P, and the tree patterns in M do not strictly subsume any of the tree patterns in P. As subsumption is used for priority, it is never possible for a tree to be of higher or equal priority than another while strictly subsuming said tree. However, if priority depends on other factors, such as tree node count, then the not strictly subsuming condition is important to ensure all tree patterns are able to be matched. Line 4 describes labeling state s with the tree pattern in M with the highest priority and line 5 adds state s as one of dfa's final states F. Line 7 and onwards describes the case where not all patterns in P are at their final state yet and should therefore further construct (prefix) tree pattern e until all non-unifying tree patterns in P have their own final state. If multiple patterns in P are of high priority and unify with one another, the highest priority between said patterns will be used for marking s as final state. Line 7 assigns c with the next tree path coordinate to match for the automaton. It does so using (prefix) tree pattern e, and all the tree patterns in P. A heuristic is used for the decision which path to check. This decision affects the automaton's construction. Because matching has to be fast, a greedy decision algorithm is used which tries to discriminate as soon as possible between all patterns in P; however, if necessary a different decision methodology can be used. Line 8 labels state s with the path string c to denote what node is matched at state s. Line 9 assigns to FS all function symbols (that are neither a variable wildcard or a constant wildcard) at tree coordinate c, for all tree patterns in P. Line 10 contains the conditional statement which decides whether the $CreateSink$ function is to be called, or a closest diverging sink state for state s is to be searched. This conditional checks whether any tree in P contains a symbol at tree coordinate c that is either a variable wildcard or a constant wildcard and if so, it will call the $CreateSink$ function using the tree patterns in P with said variable wildcard or constant wildcard at tree coordinate c. Line 11 creates a fresh state, and line 12 adds a $\neq$ sink transition to dfa which goes from s to the fresh state. Line 13 calls the $CreateSink$ function using the fresh state created in line 11, a new (prefix) tree pattern derived from e where the symbol at tree coordinate c has been replaced with the $\neq$ symbol which denotes a variable wildcard within the expression, all patterns in P which have either a variable wildcard or constant wildcard at tree coordinate c, and tree coordinate c. In reality, line 13 is a recursive call as $CreateSink$ calls the $Build$ function again after its sink creation procedure. Lines 15, 16, and 17 are the else condition for the condition on line 10. Line 15 assigns $sink$ to the closest diverging sink state for state s, line 16 is the conditional which checks whether the $sink$ variable is assigned a state or nil, where nil denotes no closest diverging sink state exists for s. Line 17, which is only invoked when the conditional on line 16 holds, creates a $\neq$ transition for dfa going from state s to state $sink$. Line 18 loops over all symbols in FS together with lines 19, 20, and 21. Line 19 creates a fresh state and line 20 adds a transition on symbol $f \in FS$ going from s to the fresh state created in line 19.

Line 21 recursively calls *Build* using the fresh state of line 19, the (prefix) tree pattern e where e's subtree at tree coordinate c has been replaced with function symbol $f \in FS$, and the set of tree patterns in P which have symbol $f \in FS$ at tree coordinate c.

Note that the lines from 10 to 17 which are responsible for the $\neq$ sink transition from state s occur before the other transitions for all symbols $f \in FS$. The reasoning for this is similar as the reasoning given for *CreateSink* where the $\neq$ transition needs to exist prior to the transition for all $f \in FS$ so that the $\neq$ transition can be used for the closest diverging sink state search for all states beneath the transitions $f \in FS$. Furthermore, all function symbols $f \in FS$ denote the not only function symbols such as *add* and *sub* which both have an arity of two, but also constant literals which are considered function symbols with an arity of zero.

Runtime complexity of this procedure is an estimated $\Omega(\sum_{p \in P} |p|)$ where $|p|$ denotes the size of pattern p (i.e. the number of nodes within the tree pattern). This runtime complexity takes some liberal assumptions:

1. Runtime of the heuristic on line 7 depends on the heuristic used and implemented; however, for it is assumed constant time $O(1)$

2. Closest Diverging Sink procedure CDS is never called, as this requires a reverse breadth first search traversal through (part of) the so far generated DFA

3. Computation of M and the final state conditional on lines 2 and 3 do not dominate the *Build* procedure's runtime complexity

4. Creation of a new state is constant time $O(1)$

5. New transitions between states is constant time $O(1)$

This runtime complexity is derived from the fact that every call to either the *Build* or *CreateSink* procedures labels the input state, creates a new state for every unique symbol at a coordinate using P, and recurses on the newly created state. Although the *CreateSink* procedure expands with an extra state, the extra states is still linearly bounded by the number of wildcard occurrances for every unique coordinate using P.

4.3.2 Back-end

The back-end phase of top-down peephole optimization matching and transforming generates C++ code for LLVM using the constructed automaton and auxiliary peephole optimization information. This section aims to explain the lowering of the given automaton and peephole optimization information into C++ code. Although C++ is the target language, alternative back-ends for different target languages can be written. Automaton and code generation exist separately which allows either to be modified without affecting the other.

Code generation

To generate code, the provided automaton is traversed using Breadth-First Search (BFS) starting at the automaton's initial state. During each traversal the code for said state is generated. Since every state with its transitions does not depend upon other states, the code generated for every state can be generated stand-alone. A state can be:

1. a regular state, or

2. a final state

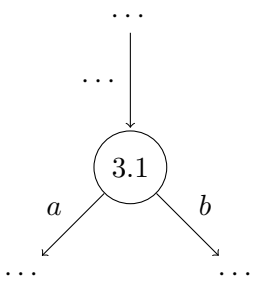


Figure 4.13: *State with transitions for symbols a and b at tree coordinate* 3.1

A regular state has transitions towards other states. These transitions can occur on symbols in Σ; however, it is possible to have states with transitions on $\neq$ sink transitions or $\neq_c$ sink transitions for constant wildcards. Each regular state is lowered into a block of code with an entry point which can be jumped towards, containing multiple conditionals that check for a match on the input tree symbol at a tree coordinate. On such a match, a jump towards the another state is made. For example, take $\Sigma = \{a, b\}$, and the state labeled 3.1 in figure 4.13. This labeled state has an underlying unique identifier in $\mathbb{N}$ to distinguish different states with the same label. For now, let this unique identifier be 1, i.e. the state labeled 3.1 in figure 4.13 is state 1. Then, the generated code will be a unique entry point which can be called or jumped towards. This entry point needs to be named in a unique and identifiable fashion, which can be done using the state's unique identifier. For state 1 in figure 4.13 this name could be 'state_1'. The contents of the code at this entry point will be conditionals which check whether a match between the input tree's symbol at tree coordinate 3.1 and one of the transition symbols occurs. In the case for 'state_1', the state will jump towards another state's entry points on when a match occurs between the input tree's coordinate 3.1 and symbols a or b. If no $\neq$ transition exists in a state, and the input tree cannot take any of the existing transitions, an implicit matching failure is assumed. For example, take $\Sigma = \{a, b, c\}$, then the state labeled 3.1 in figure 4.13 will result in a matching failure if the input tree of the automaton reaches the state labeled 3.1 while the c symbol occurs at tree coordinate 3.1. When a matching failure occurs, an implicit $\neq$ transition to a non-matching return value is generated. This can be considered a $\neq$ transition towards an additional garbage state[1] which self transitions using a $\neq$ transition. This garbage state denotes a state which will never result in an accepting state as it self-transitions on every possible input.

For a final state, the final state's peephole optimization auxiliary information which was added to the context of said final state during construction is retrieved. If an input subject tree reaches this final state, it has matched the source tree pattern of the peephole optimization associated with the final state. This means that the final state should still be an entry point that can be jumped towards, however, the contents should contain or refer to the peephole optimization's code transformation logic. Note that the peephole optimization may contain (both implicit as explicit) preconditions. In such a case an extra conditional which guards the peephole optimization's code transformation should be emitted.

[1] Also referred to as sink state in other literature

```
state_388:
{
    if (match(x_1, m_Xor(m_Value(x_1_1), m_Value(x_1_2)))) {
        goto state_406;
    } else if (match(x_1, m_NUWShl(m_Value(x_1_1), m_Value(x_1_2)))) {
        goto state_420;
    } else if (match(x_1, m_LShr(m_Value(x_1_1), m_Value(x_1_2)))) {
        goto state_399;
    } else if (match(x_1, m_Shl(m_Value(x_1_1), m_Value(x_1_2)))) {
        goto state_413;
    } else if (match(x_1, m_Trunc(m_Value(x_1_1)))) {
        goto state_427;
    } else {
        goto state_389;
    }
}
```

Listing 4.5: *LLVM C++ code example for a regular state*

A transition from a regular state to another state means gaining more information about the structure of the subject tree, used as input for the automaton. During construction of the automaton this gain of tree structure information is represented by the prefix tree that gets augmented at every recursive call which creates a fresh state. The generated code for each state also simulates this information gain by populating variables at each transition with the subtree. These subtrees can later be used in a regular state's matching or a final state's guard checking and code transforming. An example of subject tree structure information gain is shown in listing 4.5. This listing shows the C++ code contents of a state with six transitions (of which one is a fallback, sink transition). Each shown transition tries to match variable x_1 with a instruction. Within the emitted code, variables with an x prefix followed by integer values separated by underscores denote the subject tree's subtrees by their path. In this case, x_1 refers to the subtree at path 1. Not only does the condition for each transition match x_1 to one of the instructions, it also populates the children of said instructions. Take the first conditional in listing 4.5, the subtree at coordinate 1 is checked for a match on a 'Xor' instruction, and the children at coordinates 1.1, and 1.2 are populated. This also means that variable x_1 had to be populated by a state occurring prior to the state shown in listing 4.5. The only subtree populated before automaton execution is the subtree at the empty path's coordinate, denoted in generated code by variable x.

Final state expansion

Although final states denote a match for a peephole optimization's source pattern, if the peephole optimization's code transformation is guarded by conditionals then a final state is not truly a final state. The back-end code generator expands these guarded final states such that either the guard holds resulting in the code transformation, an alternative sub-automaton is transitioned towards, or if neither the guard holds and no alternative sub-automaton exists a matching failure value is returned.

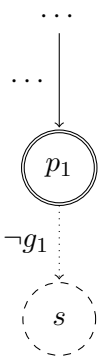


Figure 4.14: *Final state p_1 with fallback option*

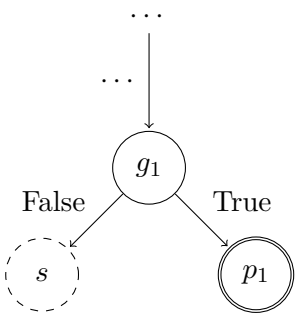


Figure 4.15: *Functional expansion final state p_1*

Figure 4.14 shows the final state p_1 as constructed by the *Build* function in algorithm 4, where the dotted transition is part of the auxiliary peephole optimization information bound to the final state. This dotted transition is the fallback transition towards sub-automata s in case guard g_1, associated with final state p_1, fails. This fallback option can be obtained using the closest diverging sink state search described by algorithm 2. If no such state is found, a dotted transition towards the blocking garbage state s is used to denote a match failure.

Figure 4.15 shows how the final state expansion should be considered, where the only responsibility of the final state is to apply the peephole optimization's code transformation. The added state g_1 is state p_1's associated guard. Outgoing transitions from g_1 are on the result of evaluating g_1. When true, a source pattern match is found and the required conditionals hold and a transition towards the final state is made. When false, the required conditionals do not hold and a transition towards the fallback sub-automata s is made.

4.3.3 Preconditions

A peephole optimization's precondition is emitted by the back-end as a guard, checked at the final state. Recall that it might be possible that no preconditions exist for a peephole optimization. In this case the procedure contents of the final state for said peephole optimization only consists of the peephole optimization's code transformation.

All peephole optimization precondition options and built-in functions require the ability to be lowered to the back-end target language's equivalent. This target language equivalent needs to check whether the precondition holds for the matched subject tree. The primary reasons why preconditions cannot be evaluated prior to source pattern matching, is because information about the subject tree is necessary. This subject tree information is retrieved while executing the automaton. When an input subject tree arrives at a final state, enough information about the subject tree has been retrieved to know its tree structure. This information is then used to evaluate the precondition, and transform the subject tree into its more efficient or canonical equivalent.

```
if (isKnownToBeAPowerOfTwo(C1)) {
    ...
    // Peephole optimization code transformation
    ...
} else {
    ...
    // Transition to fallback state, or failure return
    ...
}
```

Listing 4.6: *LLVM C++ code example for precondition*

For example, listing 4.6 contains part of the lowered C++ LLVM code of the final state for the peephole optimization defined in listing 4.2. This lowered code shows the Alive DSL built-in function 'isPowerOf2' with its subtree argument lowered to its C++ LLVM equivalent. In this final state, all necessary subtrees of the subject tree are known and ready to be used for checking whether the guard holds. If the conditional is true, the peephole optimization code tranformation is applied to the subject tree. If false, a transition is made towards a fallback state, or (if no such fallback state exists) a matching failure value is returned to break out of the automaton without applying any code transformation.

Not all preconditions are explicit, some preconditions are encoded within the source pattern tree. An example of such precondition is variable wildcard equivalence. When such implicit preconditions are detected, they are considered preconditions and only checked in final states. Having these implicit preconditions within the context of a peephole optimization's source pattern could assist in checking preconditions prior the final state. However, this will complicate automaton construction as overlapping tree patterns need to diverge in case part of the overlapping tree patterns comprise of implicit preconditions.

4.3.4 Flags

Within LLVM's IR, it is possible to add flags to instructions to augment their functionality. There are multiple approaches available to support these flags for peephole optimization matching. One is to regard these flags as additional constraints and extend the preconditions with a check for such flags. Another way is to consider flagged instructions as separate instructions and match them as such within the automaton. The difference between these two approaches is how either affects automaton construction. When considering flags as additional constraints in preconditions, matching failure on flags occur only on the final state's guard. When considering flags as part of the instruction, matching failures on flags occur during automaton execution. This means that source pattern match failures are detected earlier and unnecessary matching is preempted faster.

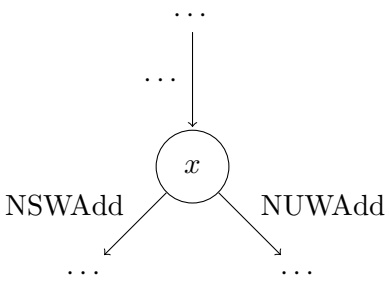


Figure 4.16: *State with separate transitions for no unsigned wrap add, and no signed wrap add*

The used approach is considering flags as part of an instruction. The resulting automaton for states with multiple transitions from flagged instructions will look like the sub-automaton shown in figure 4.16. This automaton shows a state x with separate transitions on the 'Add' instruction, depending on the associated flag. Any other instruction other than 'NSWAdd' or 'NUWAdd', including an 'Add' instruction with other flags, will result in a source pattern match failure.

4.3.5 Code flow LLVM

To integrate the top-down approach into LLVM's peephole optimizer, every available instruction node will be used as automaton input. This instruction node will be taken as root of its expression tree and put through the automaton to see if it matches any of the peephole optimization patterns. This section aims to explain the process of pattern matching, precondition checking, and the transformation of code using the top-down peephole optimization matching methodology, for an instruction node.

InstCombine

Algorithm 1 shows the naive algorithm for LLVM's instruction combine IR level peephole optimizer. The 'optimize' procedure invoked on line 8 will be the call to the top-down approach. The only provided argument is the instruction being optimized. Recall that these instructions are provided in the order that they occur within a function. This order is counter intuitive for the top-down approach as matching happens starting from the root. The provided instruction will then be used as root for top-down matching; however, this might then pass over an instruction multiple times, as part of different subexpressions.

Automaton execution

Provided to the automaton is the instruction as root of its own expression. Recall that matching occurs on transitions from one state to another. During matching all of the instruction's arguments are populated for further matching transitions. This continues until either a match failure occurs denoted by a return of a 'nullptr', or a transformation is applied. Transformations can only be applied in the final state if the precondition holds.

```
Instruction *InstCombiner::runOnInstruction(Instruction *I) {
    ConstantInt *C2, *C1;
    Value *x_1, *x_2, *x, *x_2_2, *x_2_1;
    x = I;
    goto state_0;

    ...
    // State machine
    ...
}
```

Listing 4.7: *LLVM C++ optimize example*

Listing 4.7 shows part of the procedure for top-down approach's 'optimize'. Shown is how variable 'x' is the root, initialized to the provided instruction 'I'. Note that 'state_0' is always the initial state and is, therefore, used to transition towards initially. Furthermore, note that all the 'Value' pointer variables used during matching are already defined, ready to be populated by the matching automaton.

4.4 Bottom-up

Similar to top-down peephole optimization matching and transforming, the bottom-up methodology can be considered a compiler-compiler with a traditional compiler design. The front-end that parses Alive DSL used by the bottom-up methodology is shared with Alive's formal verification of peephole optimizations. From the peephole optimization definitions in the Alive DSL, array tables are constructed and used to assign each tree node of the subject tree to a matchset, a set containing a permutation of expressions from the Pattern Forest (PF). Each of these assigned matchsets may contain source patterns part of a peephole optimization. When a matchset containing such a peephole optimization's source pattern is assigned to a node, a pattern match has been found and the peephole optimization's transformation should be applied. The array tables constructed for source pattern matching can be considered the IR for bottom-up's pattern match methodology. The constructed tables, together with the procedure that assigns matchset to a subject tree, are emitted in C++ (for use in LLVM) by the back-end.

This section aims to explain the implementation of the bottom-up peephole optimization matching and transforming methodology in Alive.

4.4.1 Methodology

For bottom-up approach's implementation, the methodology discussed by Chase in [5] is augmented and extended for the constraints introduced in section 4.1. The algorithm in [5] describes how to generate matchsets for an input pattern forest PF. Recall that PF is the set of all possible subtrees of a set of supplied patterns. For completeness, the pseudocode to generate the PF is shown in appendix A. The generated matchsets can then be enumerated and used to generate the array tables. This section aims to explain the modifications and extensions for bottom-up peephole optimization matching and transforming.

Bound wildcards

Bottom-up's approach to tree pattern matching needs to be extended to conform variable and constant wildcards. The introduction of constrained wildcards affects both PF generation and matchset generation.

Every variable or constant wildcard occurring in multiple different expressions are only bounded within its own expression and do not necessarily share the same value. This characteristic prevents unnecessary generation of variable wildcard matchsets and unnecessary subexpressions in PF. For example, take a tree pattern $add(x, y)$ where $x, y \in \mathcal{V}$, then PF will be $\{x, add(x, y)\}$, and the resulting matchsets will be:

1. $\{x\}$
2. $\{add(x, y), x\}$

Note that because of the aforementioned characteristic, PF will only contain x as variable wildcard instead of both x and y. This is because no distinction can be made between the two. Moreover, no distinction can be made between matchsets $\{x\}$ and $\{y\}$, therefore one of the two suffices to denote the matchset containing only a variable wildcard. The same holds for constant wildcards where each constant wildcard is only bound within an expression.

Matchset construction

Matchset construction is an iterative procedure which starts with an initial iteration and continues with a fixed-point iteration. The fixed-point iteration is only allowed to terminated when the iteration's resulting matchsets are the same as the matchsets generated by the previous iteration. The number of iterations of this fixed-point iteration is bounded by the maximum height, h, of all pattern trees in PF. This bound is the result of the way every iteration generates its resulting matchsets. Every iteration's result depends on last iteration's result by taking all the currently generated matchsets and check whether there exists a permutation of these matchsets used as child subtrees that results in new matchsets. Because the initial iteration starts at matchsets consisting of leaves, the second iteration consists of all matchsets consisting of leaves and trees with height = 1, and so on.

To illustrate, take the following pattern expressions:

- $add(1, add(1, x))$
- $add(0, sub(x, 1))$

Then the matchsets generated are:

1. $\{x\}$
2. $\{0, x\}$
3. $\{1, x\}$
4. $\{add(1, x), x\}$
5. $\{sub(x, 1), x\}$
6. $\{add(1, add(1, x)), add(1, x), x\}$
7. $\{add(0, sub(x, 1)), x\}$

Where $x \in \mathcal{V}$. From the generated matchsets, numbers 1, 2, and 3 are generated in the initial iterations, numbers 4 and 5 are generated in the first iteration, and numbers 6 and 7 are generated in the second iteration. An additional third iteration will execute, however, since the second and third iterations generate the same matchsets the iteration loop is terminated and all possible matchsets are discovered. Note that the maximum height h of all pattern expressions supplied is 3 which is the bound on number of iterations (leaving out the initial iteration).

Recall that every function in Σ requires its own array table. The more matchsets exists, the larger the generated function array tables. Additionally, the dimensionality of a generated table grows with respect to the function's argument count. For example, the LLVM IR 'Select' instruction requires three arguments. This results in a three dimensional array table for the 'Select' instruction. Each dimension of the naive array table has the same length as the number of matchsets. Assuming such an array table for the 'Select' instruction where every array table entry is a 32-bit integer. Then, to occupy 1 gigabyte of memory for just the 'Select' instruction's array table the number of matchsets required is 630 as $630^3 = 250.047.000$ array table elements. Then, assuming each element occupies 4 bytes resulting in $250.047.000 \times 4 = 1.000.188.000$ bytes which is about 1 gigabyte. This only worsens as the dimensionality of the generated table increases. It is therefore important that array table sizes are as minimal as possible.

As described by Chase in [5], an array table's duplicate dimensional entry (e.g. row or column in a two dimensional array table) can be omitted resulting in a minimized array table. Moreover, omitting duplicate dimensional entry can occur during matchset creation using representer sets which avoids generating the large, naive array table before minimization. These sets contain, for each function in Σ with an arity higher than 0 and for each argument number in $\mathbb{N}$, the matchsets used by function's minimized array table. Every matchset can be mapped to one in the representer set. Then, the representer set can be used to construct the minimized array table with an additional mapping step that is introduced for mapping matchsets to sets in the representer set. The size of representer sets have an upper bound equal to number of generated matchsets. In other words, the best case scenario for array table size would be when all matchsets map to a single set in the representer set whereas the worst case scenario would be when every matchset maps to a differing set in the representer set. As multiple matchsets can map to the same set in a representer set, these matchsets can be considered equivalent as their resulting mapped value is the same. For this, a notion of matchset equivalence will be introduced.

To illustrate, take the aforementioned generated matchsets example with input patterns: $add(1, add(1, x))$ and $add(0, sub(x, 1))$. The following representer sets can be generated for functions *add* and *sub*:

- representer set for *add*, 1^{st} argument: $\{\{\}, \{0\}, \{1\}\}$
- representer set for *add*, 2^{nd} argument: $\{\{x\}, \{add(1, x), x\}, \{sub(x, 1), x\}\}$
- representer set for *sub*, 1^{st} argument: $\{\{x\}\}$
- representer set for *sub*, 2^{nd} argument: $\{\{\}, \{1\}\}$

To generate these representer sets, all matchsets and child sets are required. These child sets are for each function in Σ with an arity greater than 0 and for each argument and denote all the possible subexpressions possible at a function's argument. The child sets can be computed using the following:

$$childSet_{f\in\Sigma}^{i\in\mathbb{N}} = \{s_i|\ f(s_1, \cdots, s_n) \in PF\}$$

Then, for the aforementioned generated matchsets example with input patterns $add(1, add(1, x))$ and $add(0, sub(x, 1))$, the following child sets can be generated:

- $childSet_{add}^{1} = \{0, 1\}$
- $childSet_{add}^{2} = \{x, sub(x, 1), add(1, x)\}$
- $childSet_{sub}^{1} = \{x\}$
- $childSet_{sub}^{2} = \{1\}$

Note that generation of child sets only depends on expressions in the pattern forest. Then, using the generated child sets, the representer sets can be constructed using the following:

$$representerSet_{f\in\Sigma}^{i\in\mathbb{N}} = \{r \in \{\mathcal{E}\}|\ m \in MS \wedge r = m \cap childSet_f^i\}$$

Where MS is the set of generated matchsets and r is the intersection of $m \in MS$ with a child set. The introduced matchset equivalence is when a matchset m_1 and m_2 map to the same set in a representer set, or alternatively, when $m_1 \cap childSet_f^i = m_2 \cap childSet_f^i$. This type of equivalence is referred to as the Aj equivalence [5].

Algorithm 5 Matchset generation

input:

$PF : \{\mathcal{E}\}$ - set of all subexpressions of the tree patterns

vars:

$VW : \{\mathcal{V}\}$ - set containing $v \in \mathcal{V}$ if $v \in PF$

$CW : \{\mathcal{C}\}$ - set containing $c \in \mathcal{C}$ if $c \in PF$

$iteration : [\{\{\mathcal{E}\}\}]$ - sequence of matchsets

$ms : \{\mathcal{E}\}$ - matchset used to add to iteration

$childSet_{b\in\Sigma}^{i\in\mathbb{N}} : \{\mathcal{E}\}$ - expressions that occur as the i^{th} child of patterns with symbol b in Σ

$rep_{b\in\Sigma}^{i\in\mathbb{N}} : [\{\mathcal{E}\}]$ - Sequence of sets used for representer sets

$A : \mathcal{S}$ - Function symbol for f

$C : \{\{(\mathcal{E}_1 \times \cdots \times \mathcal{E}_{ar(t\in PF)})\}\}$ - set of sets containing tuples of expressions in PF

$l : \{\mathcal{E}\}$ - new matchset to be added to current *iteration*'s matchset

$ar : \mathcal{E} \to \mathbb{N}$ - returns the number of arguments for an expression

$root : \mathcal{E} \to \mathcal{S}$ - returns the symbol at the root of an expression

1: **function** GENMS(PF)
2: $VW := \{p \in PF \mid p \in \mathcal{V}\}$
3: $CW := \{p \in PF \mid p \in \mathcal{C}\}$
4: $iteration[0] := \emptyset$
5: $iteration[0] := iteration[0] \cup \{VW\}$
6: $iteration[0] := iteration[0] \cup \{CW\} \cup \{VW\}$
7: **for all** $p \in \{o \in PF \mid ar(o) = 0 \wedge o \notin \mathcal{C} \wedge o \notin \mathcal{V}\}$ **do**
8: $iteration[0] := iteration[0] \cup \{\{p\} \cup VW \cup CW\}$
9: $childSet_G^j := \{s_j |\ G(s_1, \cdots, s_k) \in PF\}$
▷ for all $G \in \Sigma$ and $j \in \{n \in \mathbb{N} |\ 0 < n \leq k\}$
10: $rep_G^j[0] := \{r \in \{\mathcal{E}\} |\ m \in iteration[0] \wedge r = m \cap childSet_G^j\}$
▷ for all $G \in \Sigma$ and $j \in \{n \in \mathbb{N} |\ 0 < n \leq ar(G(\cdots))\}$
11: **repeat**
12: $iteration[i] := iteration[i-1]$
13: **for all** $f \in \{p \in PF \mid ar(p) > 0\}$ **do**
14: $A := root(f)$
15: $C := \emptyset$
16: **for all** $(s_1, \cdots, s_{ar(f)}) \in rep_A^1[i-1] \times \cdots \times rep_A^{ar(f)}[i-1]$ **do**
17: $C := C \cup \{s_1 \times \cdots \times s_{ar(f)}\}$
18: **for all** $t \in C$ **do**
19: $l := VW$
20: **for all** $(a_1, \cdots, a_n) \in t$ **do**
21: $l := l \cup \{A(a_1, \cdots, a_n)\} \cap PF$
22: $iteration[i] := iteration[i] \cup \{l\}$
23: $rep_G^j[i] := \{r \in \{\mathcal{E}\} |\ m \in iteration[i] \wedge r = m \cap childSet_G^j\}$
▷ for all $G \in \Sigma$ and $j \in \{n \in \mathbb{N} |\ 0 < n \leq ar(G(\cdots))\}$
24: **until** $rep_G^j[i] = rep_G^j[i-1]$ for all $G \in \Sigma$ and $j \in \{n \in \mathbb{N} |\ 0 < n \leq ar(G(\ldots))\}$
25: **return** $iteration[i], rep_G^j[i]$ ▷ for all $G \in \Sigma$ and $j \in \{n \in \mathbb{N} |\ 0 < n \leq ar(G(\ldots))\}$

Algorithm 5 shows the procedure used to generate matchsets. Lines 2 to 10 describe the initial iteration while lines 11 to 24 describe the fixed-point iteration loop for the i^{th} iteration, where $i \in \mathbb{N}$ and $i > 0$. Line 2 shows the initialization of variable wildcard set VW. If a variable wildcard occurs in the pattern forest, then VW will be the set containing a variable wildcard, otherwise VW will be the empty set. Similarly, line 3 describes the initialization of constant wildcard set CW which is the set containing a constant wildcard if one occurs in the pattern forest and will be the empty set otherwise. Line 5 shows the initialization of a matchset containing only the variable wildcard. Note that such a matchset will only exist if a variable wildcard occurs in the pattern forest. Similarly, line 6 shows the initialization of a matchset containing the constant wildcard and variable wildcard, if they occur in the pattern forest. Lines 7 and 8 contain the loop used to construct matchsets for each constant literal in the pattern forest. Recall that function symbols with an arity of zero are considered constant literals which is why the loop iterates over all expressions with an arity of zero. Line 9 constructs the child sets and line 10 constructs the representer set for the initial iteration. The loop from line 11 to 24 is the fixed-point loop that continues iterating until no more new matchset can be constructed. However, the loop from line 13 to 22 constructs the matchsets for a single iteration. Line 13 shows that only the expressions with an arity higher than zero are looped over, in other words, the constant literals are omitted. Line 14 initializes A to the function symbol of $f \in PF$. For example, if $f = add(1, x)$, then $A = add$ where $f \in \mathcal{E}$ and $A \in \Sigma$. Lines 15 to 17 construct all possible n-tuple permutations using last iteration's representer sets for symbol A. Note that the size of these tuples depend on the arity associated with symbol A. For example, if the associated arity of A is 3, then each tuple will be of size 3. These tuples are then used in the loop from line 18 to 22 which constructs new matchsets for $iteration[i]$. Line 23 simply updates the representer set for iteration i using $iteration[i]$, then the fixed-point condition on line 24 will check whether all of i's representer sets are equal the ones of iteration $i-1$. If so, all possible matchsets have been found and both the matchsets and representer sets are returned by the $GenMS$ procedure.

Matchset generation is a set heavy procedure. The dominant runtime complexity factor within the procedure are the nested loops starting at line 11. The runtime complexity estimated for these nested loops are as follows:

- line 11: bounded by maximum height h of patterns trees in PF, hence $O(h)$
- line 13: worst case, loops over all trees in PF, hence $O(\#(PF))$
- line 16: loops over the cartesian product of all representer sets rep for a function f with symbol A, hence $O(max(\bigcup_{i=1}^{ar(f)} \{\#(rep_A^i)\})^{ar(f)})$
- line 17: requires loops to compute the cartesian product of all elements in the n-tuple s (where $n = ar(f)$ and s_i is the i^{th} within the tuple) and takes the union between this cartesian product and C, hence $O(max(\bigcup_{i=1}^{n} \{\#(s_i)\})^n \times \#(C) \times max(\bigcup_{c \in C} \{\#(c)\}))$

Where # retrieves the size of a set and max retrieves the maximum value of a set. To get the estimated runtime complexity of the procedure can then be retrieved by combining the time complexities of the (nested) loops on lines 11, 13, 16, and 17.

Table construction

The function array tables can be constructed using the generated matchsets and representer sets. The representer sets can be used to construct the minimized array table. From an array table, an element can be retrieved using a tuple $(e_1, \cdots, e_n)$ as index where n is the dimension of the array table (and thereby, the arity of array table's function). The size of each dimension can be derived from the size of the dimension's representer set. For example, take the aforementioned generated representer sets example with input patterns $add(1, add(1, x))$ and $add(0, sub(x, 1))$. Then there exist two representer sets for symbol *add* because the associated arity of *add* is two. Both the representer sets contain three sets which results in a three by three array table. For the symbol *sub*, there also exists two representer sets; however, one representer set contains one set, and the other representer set contains two sets, so the array table's size will therefore be one by two. Finally, additional mapping tables are used to map each matchset to its representer set equivalent. These mapping tables are one dimensional and its sizes equal to the number of matchsets.

The elements of the table are the enumerated values that map to the matchsets. To assign these elements, the enumerated value of the matchset with the most specific pattern should be assigned to an element when the most specific pattern's child arguments' enumerated matchset values are used as the index. That is to say, when the most specific pattern in a matchset with enumerated value k is $A[x_1, \cdots, x_n]$, then the tuple $(map(x_1), \cdots, map(x_n))$ should be the index for value k, where *map* is the function that maps a subexpression to its equivalent matchset. Note that multiple matchsets may contain different expressions whose child tuple index the same element. In such a case the subsumption priority is used to determine which matchset should be assigned to the index. Recall that the subsumption priority rank more specific expressions higher.

Note that for element assignment, only the most specific pattern in a matchset is important. Because of this, a notion of reduced matchsets are used to assign enumerated values to array table elements. These reduced matchsets are the matchsets with only the most specific patterns remaining. For example, take the aforementioned generated matchsets example with input patterns: $add(1, add(1, x))$ and $add(0, sub(x, 1))$, then take the numbering as shown in the example (e.g. value 1 refers to matchset $\{x\}$, value 2 refers to matchset $\{0, x\}$, etc.). Then the following are the reduced matchsets with equivalent enumeration:

1. $\{x\}$
2. $\{0\}$
3. $\{1\}$
4. $\{add(1, x)\}$
5. $\{sub(x, 1)\}$
6. $\{add(1, add(1, x))\}$
7. $\{add(0, sub(x, 1))\}$

Computing the reduced matchset can be done as follows:

$$ms_{red} = \{p \in ms |\ \forall p' \in ms.\ p \not\geq p'\}$$

Where ms_{red} is the reduced matchset equivalent for matchset ms. The reduced matchset ms_{red} will contain only the patterns that do not subsume any of the other patterns in the matchset. The set can contain multiple trees in case these tree unify with one another.

Also note that, a 0 arity symbols in Σ (i.e. literal constants), likewise requires table generation. However, recall that the arity of a symbol affects the dimension of the array table. A zero dimension array table is merely a value, or matchset, that cannot be indexed. During matching, when a subject tree's subtree is equal to such a literal constant, the value denoting the matchset with said literal constant is assigned to the subtree. For example, take the aforementioned generated matchsets example with input patterns: $add(1, add(1, x))$ and $add(0, sub(x, 1))$. Then, the subject tree's subexpression containing the literal constant 1 will be assigned the enumerated value that maps to the matchset containing that constant literal, $\{1, x\} = 3$.

Algorithm 6 Minimized table generation

input:

$MS : \{\{\mathcal{E}\}\}$	-	all matchsets constructed by $GenMS$
$RS : \{\{\mathcal{E}\}\}$	-	all representer sets constructed by $GenMS$
$PF : \{\mathcal{E}\}$	-	pattern forest, set of all subexpressions for all matching patterns

vars:

$mapMS : \{\{\mathcal{E}\}\} \times \mathbb{N} \to \{\mathcal{E}\}$	-	returns the matchset mapped to a natural number (one natural number maps to one matchset)
$mapRS^{j\in\mathbb{N}}_{G\in\Sigma} : \{\{\mathcal{E}\}\} \times \{\mathcal{E}\} \to \mathbb{N}$	-	returns the value of a matchset's representer set
$RMS : \{\{\mathcal{E}\}\}$	-	reduced version of MS
$ar : \mathcal{E} \to \mathbb{N}$	-	gives arity (i.e. rank) of an expression
$\uparrow : \mathcal{E} \times \mathcal{E} \to \mathbb{B}$	-	unification relation of expressions

1: **function** GenTable(MS, RS, PF)
2: **if** variable wildcard $\in PF$ **then**
3: initialize all entries in *tables* with i where $x \in \mathcal{V}$ and $mapMS(MS, i) = \{x\}$
4: $RMS := \emptyset$
5: **for all** $ms \in MS$ **do**
6: $RMS := RMS \cup \{\{p \in ms \mid \forall p' \in ms \;.\; p \not> p'\}\}$
7: **for all** $ms \in RMS$ **do**
8: **for all** $t = A(t_1, \ldots, t_{ar(t)}) \in ms$ **do**
9: $\forall 1 \leq l \leq ar(t) \;.\; e_l = mapRS(RS, t_l)^l_A$
10: **if** $\exists k, j \in \mathbb{N} \;.\; \exists t' \in mapMS(RMS, k) \wedge 1 \leq j \leq ar(t) \;.\; t_j \uparrow t'$ **then**
11: $tables[A][e_1, \ldots, e_{ar(t)}] := o$ where $\{t_j, t'\} = mapMS(RMS, o)$
12: **else if** $\exists k, j \in \mathbb{N} \;.\; \exists t' \in mapMS(RMS, k) \wedge 1 \leq j \leq ar(t) \;.\; t_j < t'$ **then**
13: $tables[A][e_1, \ldots, e_{ar(t)}] := m$ where $\{t\} \subseteq mapMS(RMS, m)$
14: **return** *tables*

Algorithm 6 shows the procedure to generate a minimized array table. Supplied are the output of the $GenMS$ procedure with MS denoting all the generated matchsets and RS denoting all the generated representer sets, and the pattern forest PF. Lines 2 to 6 show the initialization necessary to assign the array table elements in lines 7 to

13. Lines 2 and 3 initialize all array table elements to the matchset containing only the variable wildcard, if one exists in PF. If no variable wildcard exists in PF, some array table elements remain empty. If during subject tree matchset value assignment an empty element is indexed, it ought to be considered a matching failure. Lines 4 to 6 reduces the matchsets in MS and creates the reduced matchset set RMS. Note that it is assumed that a matchset ms in MS and its reduced equivalent ms_{red} in RMS share the same i such that $mapMS(MS, i) = ms$ and $mapMS(RMS, i) = ms_{red}$. On line 7 starts the loop that iterates over all possible ms in RMS while on line 8 the loop that iterates over all expressions in ms start. Recall that because these are the reduced matchsets, only the most specific expressions of each matchset are iterated over. Line 9 ensures that for every l between 1 and the arity of tree t, e_l is equal to the matchset value of a matchset ms_{rs} in RMS for which holds that t_l is the argument of t, $ms \in MS$, $\{t_l\} \subseteq ms$, and $ms_{rs} = ms \cap childset^l_A$. Lines 10 and 11 cover the case when a reduced matchset exists with multiple unifying expressions that all match expression t. When this happens, the matchset value that should be prioritized is the one whose matchset covers all unifying expressions that subsume t. Lines 12 and 13 cover the case when multiple reduced matchsets exist whose expression subsumes expression t. In such a case, the reduced matchset containing the most specific expression should be prioritized for assignment. For example, the matchset $\{x\}$ subsumes every possible expression and can therefore be assigned to every element in an array table. It is more descriptive if the matchset whose value is assigned to an element contains the most specific expression as this tells most accurately the shape of the subject tree during bottom-up peephole optimization matching.

The table generation procedure is, much like matchset generation, a set heavy procedure. However, its nested loops are not as harsh as the nested loops necessary for matchset generation. The runtime complexity estimate is as follows:

- line 7: loops over all matchsets in RMS, hence $O(\#(MS))$
- line 8: loops over all trees within a matchset, hence $O(max(\bigcup_{t \in ms} \{\#(t)\}))$
- line 11,12,13,14: loops over all possible matchsets and trees within said matchset, hence $O(\#(MS) \times max(\bigcup_{t \in ms} \{\#(t)\}))$

Where $\#$ retrieves the size of a set and max retrieves the maximum value of a set. Combining these time complexities gives the estimated runtime complexity $\Omega(\#(MS)^2 \times max(\bigcup_{t \in ms} \{\#(t)\})^2)$.

Table indexing in C++

Recall that array tables have a variable number of dimensions that depend on the arity of the array table's function and that indexing is done using a n-tuple. There is no way to generalize indexing within array tables declared with multiple dimensions when these multiple dimensions differ. For example, take array tables arr_1 of size five by six (i.e. $arr_1[5][6]$) and array table arr_2 of size two by five by three (i.e. $arr_2[2][5][3]$). Both arr_1 and arr_2 contain the same number of elements; however, retrieving an element's value in arr_1 can only occur when given a 2-tuple (two indices) whereas element values in arr_2 can only be retrieved given a 3-tuple (three indices). To ensure that indexing can be accomplished in a generalized fashion, every array table is flattened to a single dimension.

The indices of the array table can then be used to compute the accumulative offset in the linearized array table. Using this methodology, calculating the to be retrieved element value's index can be done by computing the offset beforehand. This can trivially be done by looping over all indices, regardless of the number of indices. Appendx B shows the offset compute procedure for a variable number of array table dimensions.

For example, take the same arrays $arr_1[5][6]$ and $arr_2[2][5][3]$. Then, linearizing both will result in $linear_1[5 \times 6]$ and $linear_2[2 \times 5 \times 3]$ respectively. Computing the last index for both $linear_1$ (i.e. index tuple $(4, 5)$) and $linear_2$ (i.e. index tuple $(1, 4, 2)$) can be done as follows: $\underline{4} + (\underline{5} \times 5) = 29$, and $\underline{1} + (\underline{4} \times 2) + (\underline{2} \times 2 \times 5) = 29$. Note that the underlined values are the indices and the other values are the multipliers used to compute offsets.

4.4.2 Back-end

The back-end phase of the bottom-up methodology takes in the tables and mappings to emit the C++ equivalent which can be inserted into LLVM. Not only are the tables and mappings translated to a target programming language, but some additional procedures are required to traverse all tables and mappings. Finally, the peephole optimization's preconditions, flags, and transformation code have to be checked and applied. This section aims to explain the extension and decisions made for bottom-up approach's emitted code.

Mapping representation

Recall that the bottom-up approach requires mappings from a matchset's enumeration to indices of the minimized array tables. These mappings are linear, one dimensional arrays the same size as number of matchsets. Mappings exists per operand, and per instruction. This means that although every mapping array is of size $n \in \mathbb{N}$ where n is the number of matchsets, there might be multiple mapping arrays necessary per instruction since every mapping array is per dimension. For example, call to mind that the 'Select' instruction contains three arguments. The resulting mapping for the 'Select' instruction will then require three arrays, each the same size as number of matchsets.

Remember that the minimized array tables are linearized, practically flattening every array table to a single dimension. As a result the indices need to compute their offsets to retrieve the required values within this linear array table. Considering that these mappings map matchset enumerated values to the indices of the array tables, they play a role in this offset computation; however, note that during pattern matching traversal through the array tables, only the indices are used as input. The remaining values aren't affected and constant after precomputing. Because only the indices are given to the matching procedure to retrieve the enumerated matchset value, it's possible to apply the multiplier offsets within the mapping during computation of these mappings. Within the pseudocode shown in appendix B, all operations with variable m are then covered by the mapping arrays. The only operation left during bottom-up pattern matching traversal is the offset accumulation.

map_1	0	1	2	3	4
	0	1	2	1	3

$linear_1$	0	1	2	3	4
	0	1	2	1	3

map_2	0	1	2	3	4
	0	0	1	2	2

$linear_2$	0	1	2	3	4
	0	0	4	8	8

Table 4.1: *Mappings before applying multipliers*

Table 4.2: *Mappings after applying multipliers*

		First child			
		0	1	2	3
	0	0	1	2	3
Second child	1	4	5	6	7
	2	8	9	10	11

Table 4.3: *Mapping example's minimized array table*

For example, take a minimized table of size four by three (i.e. $arr[4][3]$) with the number of matchsets equal to five, and the mapping arrays for each dimension shown in table 4.1. Shown in these tables are the enumerated matchset values mapping to a particular index for the minimized array table. Because the minimized array table is two dimensional only two mapping arrays exist, each mapping differently to the minimized array table indices. When flattening the minimized array table, the mappings can be augmented to take care of any additional offsets introduced by linearizing the multi dimensional array. The tables listed in 4.2 show the same mappings with offset multipliers applied. As can be observed in table 4.2, the mapping for the second dimension has its mapping values multiplied by the size of the minimized array table. Now, to compute the correct indices for the matchsets, the mapping values are only required to be accumulated. Table 4.3 shows the minimized four by three table with each element populated with the flattened array's index equivalent. Then, using matchset enumerated values four and two as input, the following indices can be computed:

- for multi-dimensional array table: $arr[map_1(4)][map_2(2)] = arr[3][1]$
- for flattened array table: $arr[linear_1(4) + linear_2(2)] = arr[3 + 4] = arr[7]$

When using these indices in table 4.3, both the multi-dimensional and linear indices index the same element.

Note that in the aforementioned example all indices are sequential, starting at zero. It might be possible that the minimized array table does not result in sequential indices. In this case, a normalization renumbering the irregular indices to sequential indices is required.

Table representation

Every multi-dimensional array table can be linearized to a one dimensional array. For peephole optimization matching, every instruction requires one of such array tables. For example, different array tables exist for *add* and *sub* which may have different sizes. To easily find the correct array table necessary during pattern matching traversal, all

flattened tables are combined within an array used for retrieving the correct array table. Then, a mapping from instruction to natural number is introduced to easily index the right flattened table. A procedure for the target programming language is emitted which maps instructions to natural number values denoting the index of this array.

Similarly, the mapping arrays for all instructions are combined and the same mapping procedure is used to retrieve the right mapping arrays for a particular instruction.

Constants

Within bottom-up pattern matching, literal constants are considered instructions without arguments. Because a table is constructed for every instruction, every literal constant also requires an array table. Every of these literal constant array tables are zero dimension tables. This means that every literal constant directly maps to a matchset's enumerated value.

However, recall that there are also constant wildcards. If constant wildcards occur within the pattern forest, there will be a matchset representing this constant wildcard. If a subject tree's subexpression contains a constant literal not mapped to an array table, then it should be mapped to the matchset representing this constant wildcard. To support both constant literals and constant wildcards, a procedure is emitted that will return the correct matchset depending on the literal constant, including the fallback matchset containing the constant wildcard.

Found matches

When a subject tree's node has been assigned a matchset's enumerated value, it might be the case that the assigned matchset contains a peephole optimization match. If such a match occurs, the subject tree should be transformed according to the peephole optimization's transformation. Checking whether a subject (sub-)tree's enumerated value is associated with a matchset containing a peephole optimization pattern can be trivially done if the target language supports constructs such as switch cases. This switch-case would have to take in the matchset enumerated value and jumps towards transformation code depending on whether said value is a matchset that contains a peephole optimization pattern. The contents of this jump destination contains the peephole optimization's transformation in the target language. Note that the peephole optimization transformation code can still be guarded for preconditions and might still result in a match failure.

To decrease the possibility of match failures, additional jumps towards fallback transformations are introduced. These fallback transformations are according to the subsumption relation. In other words, if two peephole optimizations exists where one subsumes the other, it is possible to fallback to the more subsuming peephole optimization. The immediate subsumption graph can be used to find what peephole optimizations can be used as fallback. If no fallback exists and the precondition guard fails, a matching failure is assumed.

4.4.3 Preconditions

For bottom-up's methodology, peephole optimization preconditions (both implicit and explicit) are used as guard during transformations. This means that preconditions checks are delayed until the last moment. If such a precondition does not hold, then the subject tree does not match the peephole optimization. This approach to preconditions

is very similar to top-down's. The difference is in the fallback option. For top-down's approach, a match failure will either transition to a different sub-automaton or result in a match failure, whereas the bottom-up approach will either transition to another peephole optimization's transformation code or result in a match failure. If no precondition exists, no guard is added and the transformation will be applied the peephole optimization pattern has been matched.

Equivalence checker

Implicit preconditions relating to equivalence of subexpressions may result in an over-abundance of equivalence checks in the guard. This is the result of multiple unnecessary equivalence checks. The equivalence relation can be visualized using a graph where every vertex is a subexpression and every edge denotes an equivalence between two vertices. Then, emitting an equivelance check for every edge will result in unnecessary check because the equivalence relation is transitive. It would suffice to reduce the graph to the minimum spanning tree and only emit the remaining edges for equivalence checking.

For bottom-up's approach an equivalence checker is used to emit only the minimal number of edges necessary to denote the subexpression equivalences.

4.4.4 Flags

Instruction flags can be used to augment an instruction's functionality. To support these flags in the bottom-up peephole optimization matching and transforming, multiple approaches are available. One of such strategies is appending flag checks in the preconditions; however, using this method, the guard check can only occur at the end of peephole optimization pattern matching. This approach might complicate precondition guard predication in case the tree pattern results in different peephole optimization transformations depending on its flags (e.g. when there are both 'nsw add' and 'nuw add' patterns with the same arguments).

It has therefore been decided to concatenate flags to instructions for bottom-up pattern matching. The result is that multiple array tables may exist for an *add* instruction depending on the flags. The mapping of instructions to the correct index for the array containing array tables need to be extended to support these flagged instructions. However, this is a trivial matter as additional mappings can be added for every instruction/flag combinations to its correct array table.

4.4.5 Code flow LLVM

To match and apply peephole optimizations using bottom-up's methodology within LLVM, every LLVM IR instruction node will have to be matched, precondition checked, and, if all holds, transformed to the optimized or canonicalized alternative. However, a lot of details on how to integrate bottom-up's approach into LLVM has yet to be described. This section aims to explain the matching, precondition checking, and transforming of code within LLVM for bottom-up's strategy.

InstCombine augmentations

Algorithm 1 describes the naive instruction combine procedure. For bottom-up peephole optimization matching and transforming, a few modifications are required as a subexpression's matchset value needs to be memorized during matching. To memorize the subexpression matchset value, a hash table is used which uses expressions as key and

matchset value as its value. Within algorithm 1, the declaration of the hash table occurs between lines 4 and 5. This means that the loop starting at line 5 and iterating over elements in wl will use the same hash table keys and value, and only the fixed-point loop starting on line 2 clears hash table values at every iteration of the loop.

Then, take the 'optimize' procedure on line 8, and add the additional hash table as parameter. This 'optimize' procedure will be the bottom-up procedure for matching and transforming. Supplied is the instruction, and the hash table. Recall that the instructions supplied to this 'optimize' are in order of occurrence within the function being passed over, so every invocation of 'optimize' gets supplied the next instruction. This means that all instructions prior to the current instruction processed should already be assigned a matchset's enumerated value. Moreover, all instructions prior to the current instruction processed are either subexpressions of the current instruction processed or not used by it. This way, instructions will be supplied in the bottom-up order without additional augmentations.

Matchset value

After a matchset value for an instruction is retrieved, it is appended to the hash table using the instruction's expression as key. Note that the expressions are used as key, not only the instructions. This allows other types of expressions to be used as keys for the hash table such as expressions consisting of constant literals. Assigning an instruction's expression to a matchset value is done by taking the instruction's expression, retrieving the matchset values associated with the instruction's arguments (i.e. subexpressions) through either the hash table or a local recursive call if necessary, and using the retrieved subexpression matchset values as indices to index the element containing the correct matchset value in the instruction's array table. This matchset value can then be used by other instructions in subsequent 'optimize' calls, or can result in a peephole optimization transformation. Note that only instructions are supplied to the 'optimize' procedure. For remaining expression types required by instruction expression retrieval not yet assigned within the hash table, a recursive call for matchset retrieval is invoked. These will then be retrieved and appended to the hash table. As a fallback for matchset value retrieval, the expression could be assigned the matchset containing only the variable wildcard; however, this is only possible if a variable wildcard occurs within the pattern forest.

```
static unsigned retrieveStateValue(ConstantInt *C) {
    switch (C->getSExtValue()) {
        default: {
            return 1;
            break;
        }
        case 0: {
            return 2;
            break;
        }
        case 1: {
            return 3;
            break;
        }
        case -1: {
            return 4;
            break;
        }
    }
}
```

Listing 4.8: *LLVM C++ constant matchset retrieval*

Listing 4.8 shows an example of the matchset enumerated value retrieval procedure for constants. In this case, the constant literal values 0, 1, and -1 map to matchset enumerated values (in this example matchset enumeration values 2, 3, 4). Any other constant literal will be assigned the value for the reduced matchset containing only the constant wildcard (i.e. matchset enumeration value 1).

```
static unsigned retrieveStateValue(Instruction *I, DenseMap<Value*,unsigned>↩
     &sa) {
    unsigned offset = 0;
    unsigned child = 0;
    if(!opcodeMappingExists(I))
        return 0;
    for (Value *op : I->operands()) {
        unsigned ms = 0;
        if(!sa.count(op)) {
            ms = retrieveStateValue(op, sa);
            sa.insert(std::make_pair(op, ms));
        } else {
            ms = sa[op];
        }
        offset += computeMap[opcodeMapping(I)][child][ms];
        child++;
    }
    const unsigned *computeTable = computeTables[opcodeMapping(I)];
    return computeTable[offset];
}
```

Listing 4.9: *LLVM C++ instruction matchset retrieval*

Listing 4.9 shows an example of the matchset enumerated value retrieval procedure for instructions. Note that the matchset value 0 denotes the matchset containing only a variable wildcard. If an instruction/flag combination does not exists within the array tables, it will use the 0 matchset value as fallback. If the instruction/flag combination does exist in the array table, it will then try to retrieve its arguments' matchset values. If the instruction/flag combination does not exist in the hash table, it will attempt to retrieve them through a recursive call. Finally, the offset within the instruction's array table is calculated using each argument and finally, the right value is retrieved and returned.

Note that:

- *sa* is the hash table used
- the 'opcodeMappingExists' procedure checks whether an array table exists for a particular instruction
- the 'computeMap' array is the array with all matchset value mappings
- the 'opcodeMapping' procedure maps a particular instruction to an index
- the 'computeTables' array contains all array tables, for every supported instruction

```
static unsigned retrieveStateValue(Value *V, DenseMap<Value*, unsigned> &sa)↩
     {
    if (isa<ConstantInt>(V)) {
        return retrieveStateValue(cast<ConstantInt>(V));
    } else if (isa<Instruction>(V)) {
        return retrieveStateValue(cast<Instruction>(V), sa);
    } else {
        return 0;
    }
}
```

Listing 4.10: *LLVM C++ enumerated matchset value*

Listing 4.10 shows the procedure called when an expression's type is unknown. If the expression is a constant, it will use the procedure shown in listing 4.8. If the expression is an instruction, it will use the procedure shown in listing 4.9. If it is neither, it will use the aforementioned 0 matchset which denotes the matchset containing only a variable wildcard.

Pattern match

A pattern match is found when the matchset assigned to an expression contains a peephole optimization's pattern. Which matchsets contain such patterns can be predetermined. This allows the peephole optimizer to derive whether a peephole optimization's pattern has been matched or not by using only the matchset's enumerated value.

For example, say the expression $e = add(a, b)$ (where $a, b \in \mathcal{V}$) is a peephole optimization's pattern. Then if expression e exists within a reduced matchset, the enumerated value for e can then be used to conclude whether a subject tree matches the aforementioned peephole optimization.

When a peephole optimization match has been found, the guard predicate is evaluated and if correct, the transformation is applied. Note that during matching, the general structure of the subject tree is derived, but the specific values of the subject tree are not retrieved. Therefore, the first step for peephole optimization transformations is to retrieve the subject tree expressions. These can then used for either precondition checking or code transforming.

Chapter 5

Evaluation

The initial implementations of both top-down and bottom-up are benchmarked together with Alive's naively generated C++ code, and LLVM's default InstCombine by compiling provided benchmarks in the LLVM test-suite[1] and SPEC2017[2]. These four versions are then compared for compilation time, execution time, instruction memory size, and number of peephole optimizations applied. Note that the differences between the four versions is only the peephole optimizer. The differences between peephole optimizers are as follows:

- LLVM's default InstCombine, LLVM IR level peephole optimizer as shipped with LLVM
- Alive naive code generator, Alive's default code generator which uses the same matching strategy as LLVM's default InstCombine
- Top-down Alive code generator, implementation for Alive's top-down strategy
- Bottom-up Alive code generator, implementation for Alive's bottom-up strategy

These will be benchmarked using the same version of LLVM with the utilized peephole optimization strategy as only modification. At the time of writing, the most recent LLVM version released is 7.0.1 which will be used for benchmarking. Additionally, Alive's default, top-down, and bottom-up code generators support about 414 peephole optimizations whereas LLVM's InstCombine manages more peephole optimizations; however, the precise number of peephole optimizations is unknown. This difference in number of peephole optimizations available may affect the comparison between the Alive-based strategies, and LLVM's default InstCombine. Additionally, the impact of runtime complexity on the actual runtime is evaluated together with the resulting intermediate formats of both top-down and bottom-up.

5.1 Strategy evaluation

Both top-down and bottom-up peephole optimization matching strategies generate an immediate format. In top-down's case this format is the DFA constructed, in bottom-up's case these are the matchsets and function array tables created. This section aims to evaluate the impact of each strategy's runtime complexity on actual generation time, and the size of the generated immediate format.

[1] https://github.com/llvm/llvm-test-suite
[2] https://www.spec.org/cpu2017/

Top-down

Recall that the top-down strategy generates a DFA. When supplying Alive with the 414 peephole optimizations available for code generation, the current top-down code generation implementation results in a DFA of 1892 states and 2776 edges. Augmentations on input, heuristic used within the top-down methodology, and priority used affects the size and, more importantly, the shape of this DFA.

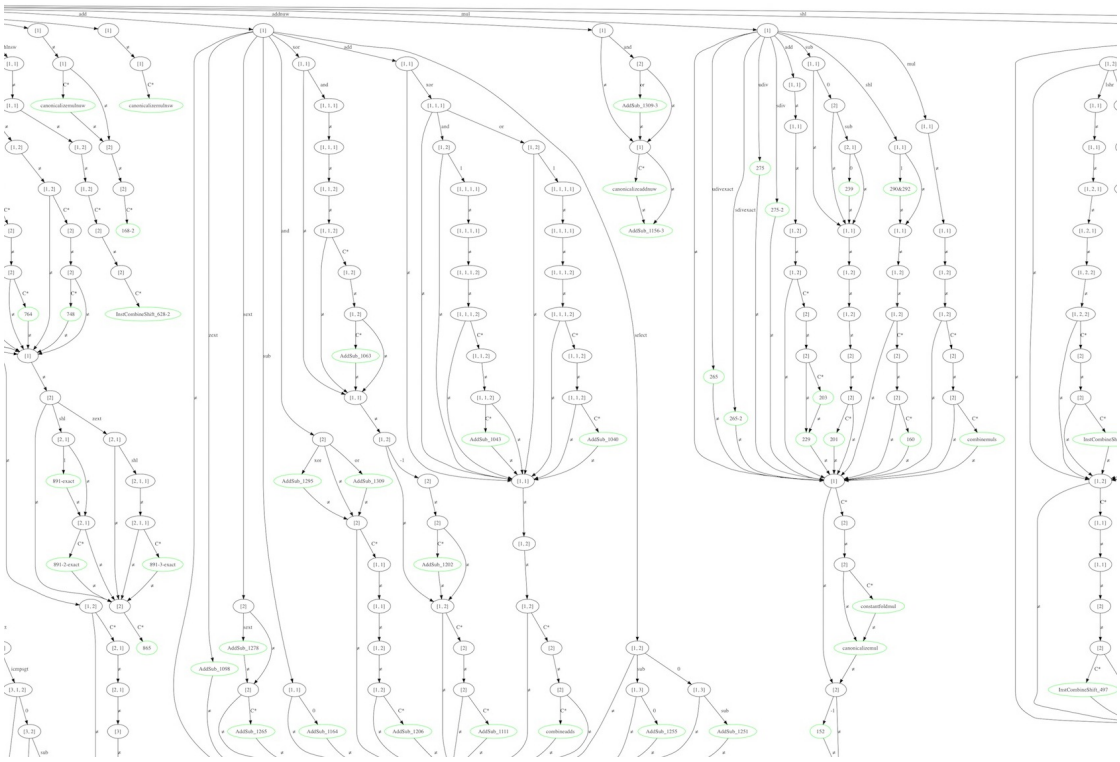


Figure 5.1: *Part of top-down DFA visualized*

Figure 5.1 shows part of the DFA constructed when the top-down code generator is provided the aforementioned 414 peephole optimizations. All states share the same initial state at the top but diverge. All states with a green outline denote a final state. Recall that transitions going outward from any final states get expanded during code emitting.

The runtime complexity for top-down's automaton construction described in chapter 4 is derived using an optimistic scenario; however, even with added complexity from additional calls (such as CDS) results in a feasible runtime using a modern computer (at the time of writing). Currently, when provided with the set of 414 peephole optimizations, the generation time for the automaton is about 2 minutes.

Bottom-up

Bottom-up requires generation of matchsets and array tables. Recall that these array tables are minimized and linearized. Moreover, they require additional mapping arrays. Displaying all array tables generated using the 414 peephole optimizations is not feasible; therefore only the size of each table is summarized and shown. The current bottom-up procedure generates 1399 matchsets when given the aforementioned 414 peephole

optimizations. The size of the mapping arrays for each instruction is merely the total number of matchsets multiplied with the arity of the instruction and these sizes are, therefore, omitted.

Instruction	and	lshr	shl nuw	select	shl nsw nuw
Array table size	46×31	12×3	3×2	$39 \times 23 \times 16$	1×2

Instruction	xor	sub	mul nsw	ashr exact	mul nuw nsw
Array table size	28×14	9×24	2×2	2×2	1×2

Instruction	mul nuw	shl nsw	add	mul	icmp ne
Array table size	2×2	2×3	19×9	13×4	2×3

Instruction	trunc	icmp slt	lshr exact	sub nsw	srem
Array table size	5	5×3	2×2	2×3	1×3

Instruction	icmp ult	sext	urem	icmp ule	icmp eq
Array table size	2×3	3	3×3	2×2	4×3

Instruction	sdiv exact	icmp sge	udiv exact	sdiv	icmp ugt
Array table size	3×2	2×2	4×6	8×5	1×3

Instruction	add nsw	icmp sgt	add nuw	udiv	zext
Array table size	3×2	5×4	3×2	10×8	3

Instruction	shl	or	ashr	icmp uge
Array table size	21×3	38×44	8×3	3×2

Table 5.1: *Array table sizes*

Table 5.1 show the array table sizes for each supported instruction. Note that the 'select' instruction contains the most number of elements with $39 \times 23 \times 16 = 14352$. Recall that this is after reduction as the complete array table would require $1399^3 = 2738124199$ elements.

The estimated time complexities for bottom-up's approach introduced in chapter 4 are described for two separate procedures. One, for matchset generation and another for table generation. On a modern computer (at the time of writing) the matchset procedure takes roughly 5 hours to compute all matchsets given the 414 peephole optimizations whereas table generation takes 6 minutes but assumes all matchsets readily available. This is roughly as expected as the estimated runtime complexity for matchset generation was derived from needing many set based operations including multiple cartesian products while table generation merely iterates through the generated matchsets, and trees within said matchsets.

5.2 Benchmarks

The LLVM test-suite contains a plethora of applications (such as ALAC, SPASS, aha, lua, oggenc, sqlite3), tests, and benchmarks (such as Adobe's C++ benchmarks, Dhrystone, Linpack, Polybench, MiBench, Rodinia, SciMark2, and many more benchmark suites commonly used for publications). In addition to these benchmarks supplied with LLVM's test-suite, the SPEC2017 benchmarks have been added to further benchmark and compare all versions of LLVM's peephole optimizer.

The benchmarks run as follows:

1. Single-core clang with bottom-up/top-down/alive-default/llvm-default peephole optimizer compiles all benchmarks

2. Compiled programs are executed and tested using test scripts

Step 1 results in compilation time, number of peephole optimizations applied and instruction memory size while step 2 results in execution time.

Note that a vectorization pass (slp vectorizer) has been disabled to avoid assertion errors in the Alive-based peephole optimization strategies. For fair comparisons, this vectorization pass has been disabled for all strategies which may affect some of the benchmark results.

Number of instructions combined

This section explores the number of peephole optimizations matched and applied (i.e. instructions combined) for each of the peephole optimization version.

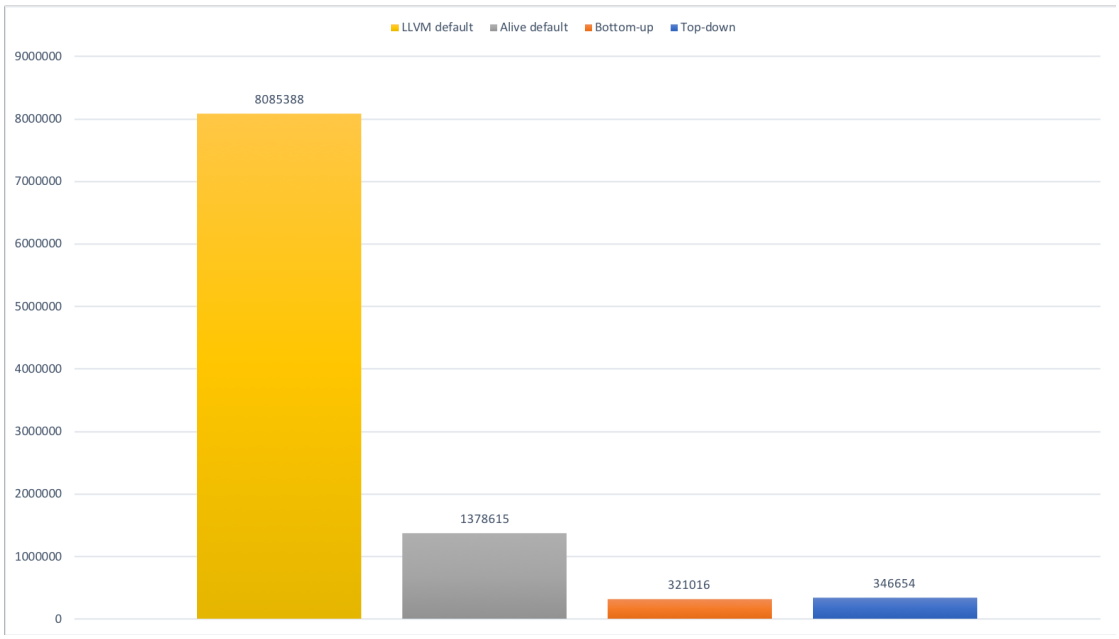


Figure 5.2: *Sum total number combined instructions, per version*

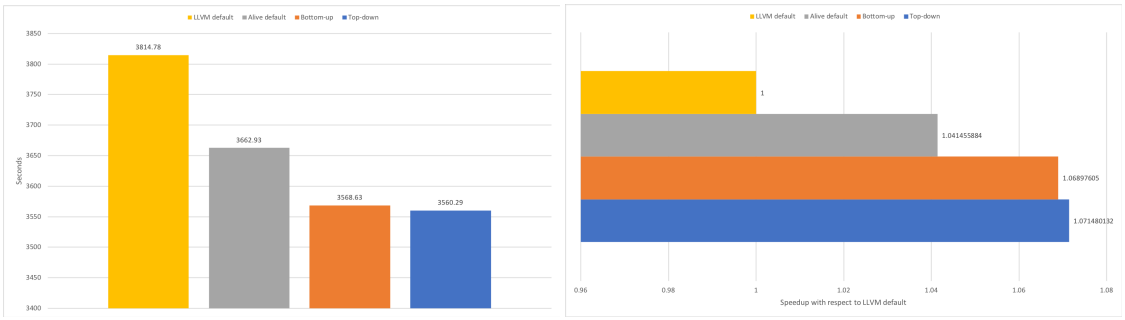
Figure 5.2 shows the total number of instructions combined for each version. In other words, the number of instructions combined for every benchmark have been accumulated and visualized per version. Observe from figure 5.2 that the default LLVM peephole optimizer combines most instructions at $8,085,388$ which is more than all of the Alive-based

peephole optimizations applied, combined. Alive default has the second most combined instructions at $1,378,615$ combines which is over a million more than either bottom-up and top-down. The difference between top-down and bottom-up is less distinct with top-down applying $346,654$ combines and bottom-up applying $321,016$ combines.

The ratios of instructions combined between the versions is an accurate representation of each benchmark's results where LLVM default generally dominates with the number of instructions combined, Alive default second most followed by either top-down or bottom-up. For example, the 'parest' benchmark part of the SPEC benchmark set shows the most number of instructions combined by any of the variants with $1,282,360$ instructions combined by LLVM default, $221,978$ by Alive default, $59,390$ by top-down, and $55,571$ by bottom-up. The only benchmarks that do not follow this kind of trend are tests with very few (< 100) instructions combined by any of the versions. For example, the 'lowercase' benchmark part of the LLVM test-suite in 'SingleSource/Benchmarks/Misc/lowercase' shows 30 instructions combined by LLVM default, 31 by Alive default, 21 by top-down, and 18 by bottom-up.

Compilation time

This section will evaluate compilation time which denotes the time (in seconds) it takes to compile a particular benchmark.



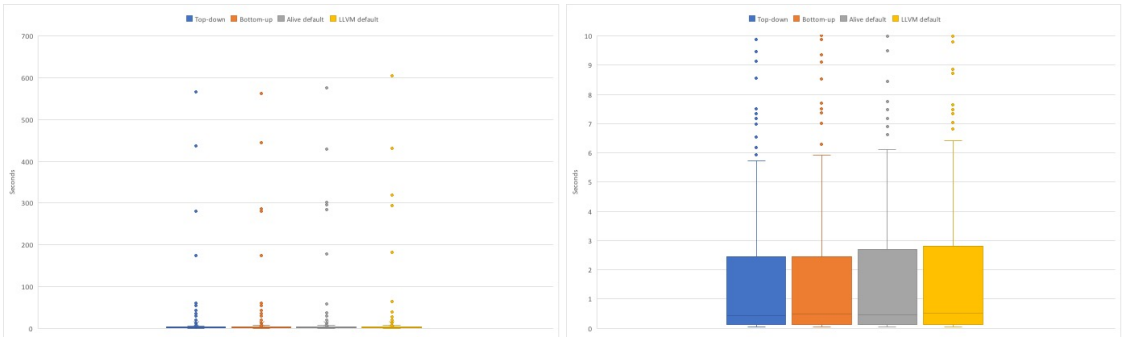
(a) *Total time in seconds to compile all benchmarks, per version*

(b) *Compile time speedup with respect to LLVM default*

Figure 5.3: *Compilation time and associated compilation speedup*

Figure 5.3a shows the total compilation time, in seconds, required per version to compile all available benchmarks with figure 5.3b showing the corresponding speedup of all four methodologies with respect to LLVM default. The compilation times shown in figure 5.3a displays an expected trend given figure 5.2 which shows that most code transformations are applied by LLVM default, then Alive default with top-down and bottom-up versions the least number of transformations. As LLVM default and Alive default apply most tranformations, their compilation time take the longest. LLVM default takes 3814.78 seconds in total, Alive default takes 3662.93 seconds in total, with top-down taking 3560.29 seconds and bottom-up taking 3568.63 seconds. These results align with the expectation that more transformations applied means more time spend doing peephole optimizations.

The corresponding speedups depicted in figure 5.3b show a (rounded to 3 decimal places) speedup of 1.000 for LLVM default showing that this is used as reference, 1.041 for Alive default, 1.069 for bottom-up, and 1.071 for top-down.



(a) *Full boxplot of compilation time* ***(b)*** *Partial boxplot of compilation time*

Figure 5.4: *Boxplots of compilation time*

Figures 5.4a and 5.4b show the boxplots for compilation time with figure 5.4b showing the same boxplot as 5.4a zoomed in until 10 seconds. Note that the boxplots omit all unit tests of the benchmarks since these take close to 0 seconds for any variant to compile, hence is not of significance for the boxplots. The remaining benchmarks are still dominated by low compilation time; however, many outliers exist in the boxplots' upper whisker as shown in figure 5.4a. LLVM default has 47 upper outliers, Alive default has 46 upper outliers, bottom-up has 46 upper outliers and top-down has 48 upper outliers with extremes going up to 604.27 seconds for LLVM default, 576.18 seconds for Alive default, 562, 87 seconds for bottom-up, and 565.70 seconds for top-down.

The contrast in compilation time between the Alive-based versions and LLVM default is not as distinct as the difference in number of instructions combined. Possibly because the minor differences between each methodology's implementation. Recall that the only difference between all versions benchmarked is the peephole optimizer whereas the remaining optimizations applied (apart from slp vectorization, which has been disabled for all) are still applied. The perceived difference in compilation time come from the number of instructions combined by the peephole optimzer, the strategy used to do so, and/or the propagated effects of the peephole optimizer on other optimizations.

Note that despite having the least number of instructions combined, the bottom-up strategy results in more time spend compiling than the top-down methodology. This does not necessarily dictate that the top-down methodology for peephole optimization matching is better than bottom-up's but that within the current implementations of either top-down and bottom-up, top-down is more successful at number of combines and compilation time. This is not entirely surprising as more time was spend on top-down's implementation whereas shortcomings and bottlenecks within bottom-up's approach might still contain trivial solutions.

Execution time

All benchmarks run tests using the compiled programs which allows evaluation of execution time. In other words, the impact of different peephole optimizer versions on execution time can be assessed. Note that LLVM test-suite's microbenchmarks, unit tests and regression tests are omitted. Microbenchmarks are run iteratively until measurements are of statistical significance. The number of iterations might differ per version and is, therefore, not suitable for a fair comparison between the different versions. Additionally, the unit and regression tests are too small for consideration. Many of the execution times of these tests are rounded to two decimal places resulting in 0.00, implying an execution

time of 0.

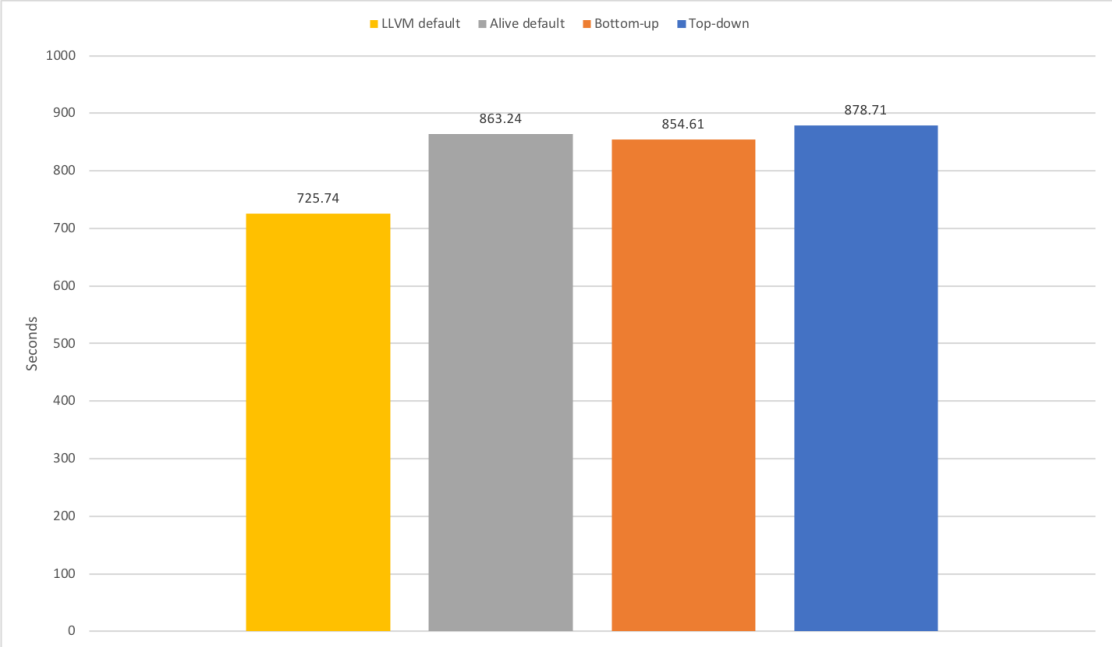


Figure 5.5: *Total execution time, per version*

Figure 5.5 depicts the total execution time, in seconds, per version. LLVM default is, as expected, the fastest with 725.74 seconds. This is most likely the result of the substantial amount of instructions combined as can be observed in figure 5.2. Surprisingly, of the Alive-based methodologies, bottom-up shows the fastest execution time with 854.61. Alive default was the expected best between the Alive-based approaches considering the number of instructions combined (which then resulted in a higher compilation time) but resulted in a total execution time of 863.24 seconds. Although top-down is the slowest in terms of execution time at 878.71 seconds, it shows the highest speedup in figure 5.3b.

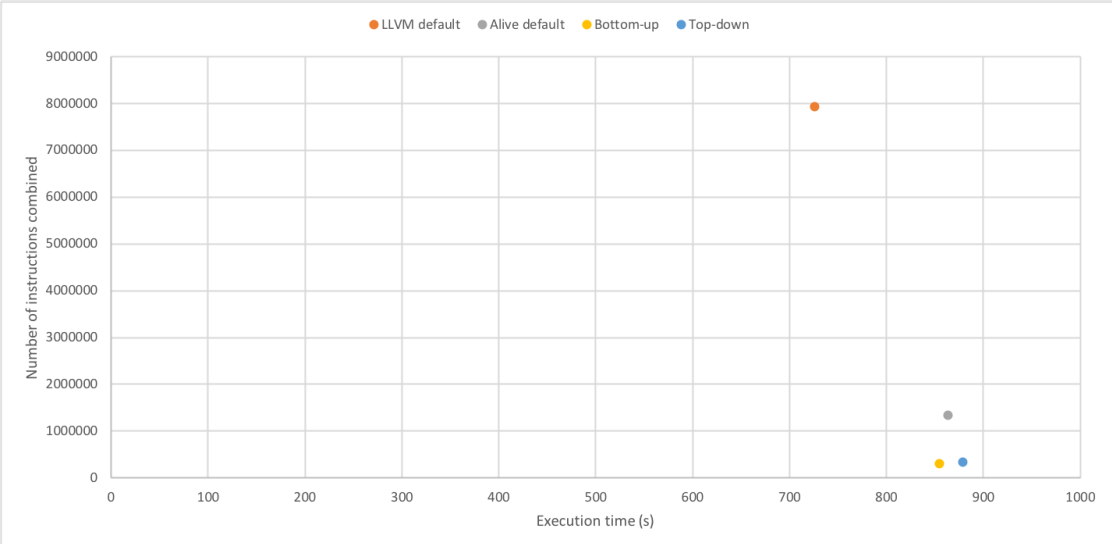


Figure 5.6: *Execution time vs number of instructions combined, per version*

Figure 5.6 shows number of instructions combined on the y-axis and execution time on the x-axis. Observe that LLVM default, Alive default, and top-down all show that a low number of instructions combined comes at a cost of execution time; however, bottom-up shows that a lower number of instructions combined can result in a lower execution time (relative to top-down and Alive default). This event may be caused by either the subsumption priority enforced resulting in better choice of peephole optimization, the matching methodology, or a combination of both.

Instruction memory size

The compiled programs can be examined for their executable instruction size (i.e. the size of the '.text' section). With this, the impact of the different peephole optimization verions can be evaluated for resulting instruction memory size.

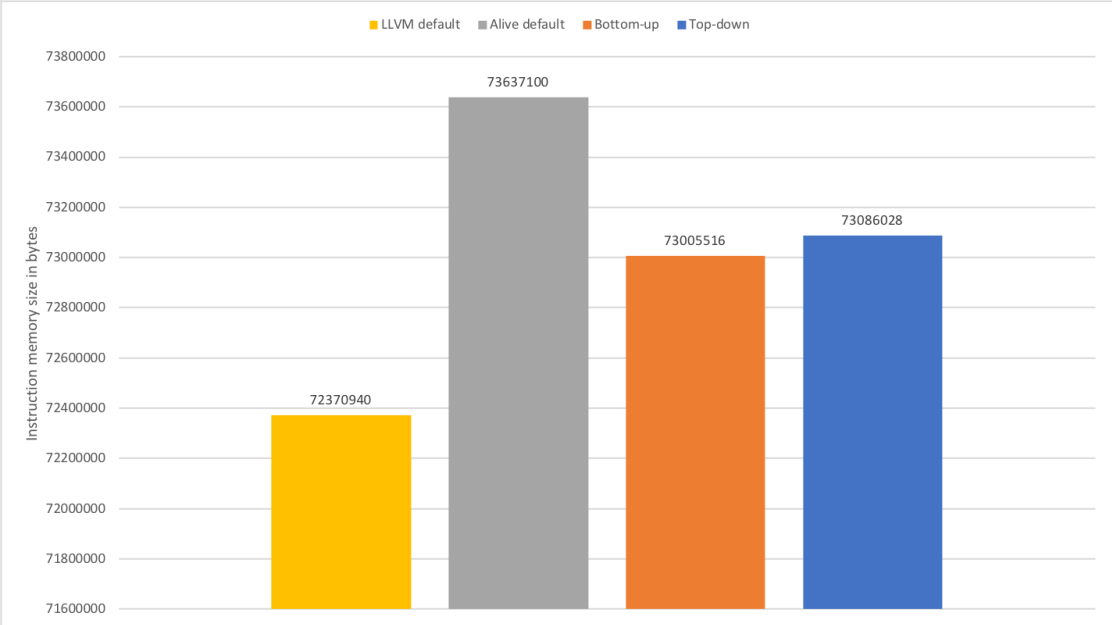


Figure 5.7: *Total size of instructions emitted in bytes, per version*

Figure 5.7 shows the total size of instructions emitted, per variant. The sizes do not deviate more than $1,266,160$ bytes which shows how evenly matched the different peephole optimizer versions are in terms of emiting instructions. Not entirely surprising is the LLVM default version which shows the smallest size of emitted instructions at $72,370,940$ bytes. The Alive-based approaches show a similar trend as with execution time, which is that Alive default emits most instructions with $73,637,100$ bytes whereas top-down and bottom-up are more conservative in terms of emitting code with $73,086,028$ and $73,005,516$ bytes emitted, respectively. This, again, shows that the number of instructions combined does not necessarily indicate better resulting executables.

Chapter 6

Conclusions

This report presented extensions to two tree pattern matching methodologies to conform to constraints introduced by peephole optimizations. One methodology was top-down, which matches trees starting from the top, going down towards its leaves. The other methodology was bottom-up, which matches trees starting from the bottom, going up towards its root.

An initial implementation of a code generator for both top-down and bottom-up methodologies have been implemented in Alive, whose DSL shows an abstract and high-level representations of peephole optimizations with the original intent to formally verify said peephole optimizations for the LLVM compiler suite. This allowed lowering of high-level representations of peephole optimizations to code which exploits the tree pattern matching methodologies. This code can then be easily included in LLVM, resulting in LLVM using said tree pattern matcher for its peephole optimizer with as goal a faster compilation time.

The initial benchmark results show promise, where both top-down and bottom-up approaches exhibit compilation time speedups compared to both LLVM default and Alive default. Moreover, the expectation was that a speedup would come at a cost of either binary size and execution time of the programs compiled using compilers which utilize tree pattern matching. This seemed to be true to some extend with LLVM default which showed the highest compilation time, most number of instructions combined, but lowest execution time and smallest instruction memory size. However, between the Alive-based methodologies the Alive default stategy showed the highest compilation time and most number of instructions combined did not show the lowest execution time or instruction memory size. Instead, the top-down strategy showed the smallest instruction memory size whereas the bottom-up strategy showed the lowest execution time. Assuming that the only factor affecting benchmarking were the differences between these four peephole optimization stategies, the compilation speedup might be caused by the subsumption priority introduced and enforced, the tree pattern matching strategy, the propagated effects of the different peephole optimizers on other optimizations, or a combination of the aforementioned.

However, despite showing an improvement using both top-down and bottom-up approaches, the differences between the Alive-based are not too notable. For example, the difference in instruction memory size between Alive default and the top-down approach is $631,584$ bytes (i.e 632 kilobytes) for the benchmark sets totaling about 72 megabytes. Moreover, the difference in execution time between Alive default and bottom-up is 8.63 seconds for a total of about 860 seconds. Neither these statistical differences seem that noteworthy, even when considering that only a variantion in peephole optimizer caused

this difference.

6.1 Open issues

This section aims to introduces some of the open issues within both top-down and bottom-up approaches to peephole optimization matching and the code generation required for it.

Supported instructions

Not all available LLVM IR instructions are supported by the code generator. Currently, these instructions are regarded as wildcard variables; however, if a peephole optimization definition which uses these unsupported instruction is supplied to Alive in the Alive DSL, with the intention of emitting LLVM C++ code, no code will be generated or emitted. The following LLVM IR instructions are supported by both bottom-up and top-down methodologies at the time of writing: **add, sub, mul, udiv, sdiv, urem, srem, shl, ashr, lshr, and, or, xor, trunc, zext, sext, ptrtoint, inttoptr, bitcast, icmp, and select**. Remember that these are the instructions supported for pattern trees. All instructions are supported as part of subject trees, as other types of instructions are covered by variable wildcards.

One of the unsupported instructions that is considered an edge case is the 'GetElementPtr'. Most instructions have a fixed number of arguments; however, the 'GetElementPtr' instruction is one of the instructions that does not have a fixed number of arguments. This affects how the tree pattern matching methodologies create its automaton or tables as these depend on the number of arguments. For example, bottom-up's approach creates a table per instruction with each table's dimension equal to the number of arguments for said instruction. As the 'GetElementPtr' instruction does not have a bound on number of arguments, the 'GetElementPtr' table would have no bound on its dimensionality. Furthermore, since the number of arguments are variable, it would either require the 'GetElementPtr' table to support multiple indices, or multiple 'GetElementPtr' tables, one for every requires number of arguments.

Priority for unifying trees

It might be possible that multiple patterns match a particular subject tree. In such a case, the priority as introduced in section 4.2 is used to derive the highest priority pattern. Nevertheless, if these tree patterns unify no priority can be derived. Topological sort using the immediate subsumption graph has arbitrarily assigned one tree pattern a higher priority than the other, however, this higher priority tree pattern might not be the most suitable choice.

Moreover, when a tree has been chosen between multiple options, the other is not considered as fallback option. This scenario may result in no peephole optimization applied when it may have been able to. For example, if a peephole optimization's precondition does not hold it will then consider all tree patterns subsuming it as fallback, but not the unifying tree.

Overlapping tree pattern structures

Multiple peephole optimization definitions might share the same tree patterns with different constraints. Recall that constraints are translated into guards for the transfor-

mation. Preferably, a tree pattern match is allowed to contain multiple conditionals to different transformations. This way, peephole optimizations with overlapping tree pattern structures can be supported.

Array table empty elements

Tree patterns can exist without wildcards. When supplying the bottom-up strategy with a set of peephole optimizations containing no variable wildcard, the generated tables might contain empty index values. In other words, some indices might result in no matchset value assignment. Indexing to these empty elements should result in matching failures; however, bottom-up's LLVM C++ generation does not support these empty elements yet and always assumes that variable wildcard occur within at least one of the supplied tree patterns.

Equivalence checker for top-down

If a tree pattern contains a plethora of implicit equivalences, many unnecessary equivalence checks will be appended to the precondition. Section 4.4.3 contains the 'equivalence checker' section which explains how unnecessary equivalence checks are omitted from the precondition. This 'equivalence checker' is not yet implemented for the top-down methodology.

Constant value assumptions in LLVM

C++ code listing 4.8 shows the matchset value retrieval function for constant literals (i.e. function symbols with an arity of 0). This function uses the 'getSExtValue' function, part of the 'ConstantInt' class. This function returns a signed 64-bit value, meaning that it assumes the value is at most a signed 64-bit. Values wider than 64-bits have not been tested.

Compare instruction predicate wildcards

Neither bottom-up, nor top-down have been tested with predicate wildcards for integer compare instructions. In other words, it is always assumed that integer compare instructions have explicit predicates (e.g. 'icmp eq' or 'icmp ule'). Predicate wildcards could be processed much like constant wildcards, where predicate wildcards subsume explicit predicates.

Test failures

Although all benchmark programs were compiled some top-down experienced a test failure on one of the benchmark sets. More specifically, the executable program emitted for the 'siod' benchmark was tested for a particular input for which the output did not match the expected output while no tolerances were allowed.

Moreover, the slp vectorizer resulted in assertion errors when used in combination with any of the Alive-based peephole optimizer. The initial debugging showed that a 'GetElementPtr' instruction contained an illegal type discrepancy between indices. This might have been initially caused by a peephole optimization and propagated through other optimizations; however, due to time constraints it has been decided to disable the slp vectorizer instead for benchmarking.

6.2 Future work

This section aims to investigate some to be explored areas within peephole optimization matching and transforming.

Alternative automata

So far, only augmented DFAs are used for construction. However, not all peephole optimization properties are matched yet during automaton traversal (e.g. preconditions). Using other types of automata with different characteristics might help matching some of these properties. For example, it might be possible to exploit the auxiliary stack supplied in push-down automata for matching some of the peephole optimization's characteristics.

Commutativity

Some instructions are commutative, meaning the argument order supplied does not affect the outcome. An example of such instruction is the 'add' instruction. Doing an $add\ 5, 2$ or $add\ 2, 5$ instruction result in 7, hence argument order does not matter. The current peephole optimizer explicitly matches subexpressions and depends on canonicalization for its matching. For example, $add\ C, x$ where $x \in \mathcal{V}$ and $C \in \mathcal{C}$ will ensure that the constant literal occurs on the left subexpressions and anything else may occur on the right subexpression. Canonicalizing the order ensures that the order between arguments, however, this will require an additional transformation. For example, if the 'add' instruction canonicalization requires constants to be the right subexpression whereas anything else is required to be on the left subexpression, then the aforementioned pattern $add\ C, x$ requires the intermediate transformation $add\ C, x \rightarrow add\ x, C$ before any further peephole optimization can be applied. Another commutativity strategy would be to match all argument permutations of commutative instructions; however, this will expand top-down automaton's state space and bottom-up's matchsets drastically.

Both discussed strategies come with their advantages and disadvantages. Preferably, commutativity can be considered during matching without any additional steps. This might be possible by, for example, constructing an alternative automaton which contains auxiliary memory to keep track of matched subexpressions.

Precondition guard offloading

Checking whether preconditions hold is currently delayed until the end of tree pattern matching. Preferably these are covered during matching of trees. This way, if preconditions are found to not hold for a subject tree, a fallback alternative or matching failure can occur faster.

Matchset generation

Currently, matchset generation for bottom-up's methodology requires a lot of time. This is because of an excessive amount of set operations. Preferably, another procedure is used to construct the matchsets. For example, by extending the table generation procedure by Hoffmann and O'Donnell [8] to support more than simple pattern forests.

Additional InstCombine improvements

Although the focus of this thesis was using tree pattern matching to speed up compilation time, other strategies to speed up the peephole optimizer exist. For example, decreasing unnecessary fixed-point iterations, preferably removing the fixed-point iteration loop altogether.

Top-down conditional branching vs switch-case jump table

Shown in listing 4.5 is a state in top-down's approach. Conditional branching is used to determine which state to transition towards. Instead of using conditional branches which require evaluation, a switch-case might be more suitable. This switch-case will not require evaluating all expressions and, instead, creates a jump table which jumps towards the next 'goto' transition to the next state. Additional investigating is required to analyse the impact of conditional branching versus a jump table within top-down's implementation.

Top-down automaton passes

Top-down approach's generated automaton still emits an inefficient automaton. For example, section 4.3 describes the 'CreateSink' procedure which expands to multiple states that might result in unnecessary states. Additionally, no state space minimization is applied for the generated automaton. These inefficiencies can be optimized by doing a pass over the automaton to analyse, detect, and optimize where possible.

Transformation occurrence

Peephole optimization transformations now occur immediately when a match has been found; however, it may be more efficient to iterate multiple times using the peephole optimizer before applying any transformation, and merging the transformations found. This could avoid unnecesary intermediate transformations.

Bibliography

[1] Static single assignment book, 5 2018.

[2] A. V. Aho and M. J. Corasick. Efficient string matching: An aid to bibliographic search. *Commun. ACM*, 18(6):333–340, June 1975.

[3] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[4] J. Berstel, L. Boasson, O. Carton, and I. Fagnot. Minimization of automata. *CoRR*, abs/1010.5318, 2010.

[5] D. R. Chase. An improvement to bottom-up tree pattern matching. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, pages 168–177, New York, NY, USA, 1987. ACM.

[6] E. de Vink. Chapter 2; finite automata and regular languages, 2 2016.

[7] J. Freire, D. Koop, E. Santos, C. Scheidegger, C. Silva, and H. T Vo. The architecture of open source applications, 01 2011.

[8] C. M. Hoffmann and M. J. O'Donnell. Pattern matching in trees. *J. ACM*, 29(1):68–95, Jan. 1982.

[9] N. P. Lopes, D. Menendez, S. Nagarakatte, and J. Regehr. Practical verification of peephole optimizations with alive. *Communications of the ACM*, 61(2):84–91, Feb. 2018.

[10] R. Nederpelt and F. Kamareddine. *Logical reasoning : a first course*. Texts in computing. King's College Publications, 2004.

[11] N. Nedjah and L. Mourelle. Optimal adaptive pattern matching. 2358:768–779, 06 2002.

[12] N. Nedjah, C. D. Walter, and S. E. Eldridge. Optimal left-to-right pattern-matching automata. In M. Hanus, J. Heering, and K. Meinke, editors, *Algebraic and Logic Programming*, pages 273–286, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[13] R. C. Sekar, R. Ramesh, and I. V. Ramakrishnan. Adaptive pattern matching. In W. Kuich, editor, *Automata, Languages and Programming*, pages 247–260, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg.

[14] K. Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, June 1968.

[15] G. van Noord. Treatment of epsilon-moves in subset construction. In *Proceedings of the International Workshop on Finite State Methods in Natural Language Processing*, FSMNLP '09, pages 57–68, Stroudsburg, PA, USA, 1998. Association for Computational Linguistics.

Appendices

Appendix A

Pattern Forest pseudocode

Procedure used to generate the Pattern Forest. Iteratively adds all expression patterns with its subexpressions to the pattern forest PF.

---

input:

$P : \{\mathcal{E}\}$	-	set of matching pattern expressions

vars:

$T : \{\mathcal{E}\}$	-	set of patterns to be added to pattern forest
$PF : \{\mathcal{E}\}$	-	pattern forest, set of all subexpressions in P
$children : \mathcal{E} \to \{\mathcal{E}\}$	-	given an expression, returns all child expressions

1: **function** GenPF(P)
2: $PF := \emptyset$
3: $T := \{p \in P\}$
4: **for all** $t \in T$ **do**
5: $T := T / \{t\}$
6: $PF := PF \cup \{t\}$
7: $T := T \cup children(t)$
8: **return** PF

Appendix B

Linear array offset computation

Procedure used to calculate the offset in a flattened, or linearized, array which was previously multi-dimensional. Effectively converts indices in *indices* for the multi-dimensional array table to a single index for the linearized equivalent array table.

input:

$table : [\mathbb{N}]_1 \times \cdots \times [\mathbb{N}]_n$	-	multi dimensional array table of size n
$indices : \mathbb{N}_1 \times \cdots \times \mathbb{N}_n$	-	tuple with n indices where n is the number of array table dimensions

vars:

$m : \mathbb{N}$	-	multiplication offset for table
$offset : \mathbb{N}$	-	accumulative sum of all indices' offsets
$size : \mathbb{N}_1 \times \cdots \times \mathbb{N}_n \to \mathbb{N}$	-	Size of tuple
$dimension : [\mathbb{N}]_1 \times \cdots \times [\mathbb{N}]_n \times \mathbb{N} \to \mathbb{N}$	-	retrieves a table's dimension's size

```
1: function OFFSETCOMPUTE(table, indices)
2:     m := 1
3:     offset := 0
4:     for all idx := 1 → size(indices) do
5:         offset := offset + (indices[idx] × m)
6:         m := m × dimension(table, idx)
7:     return offset
```