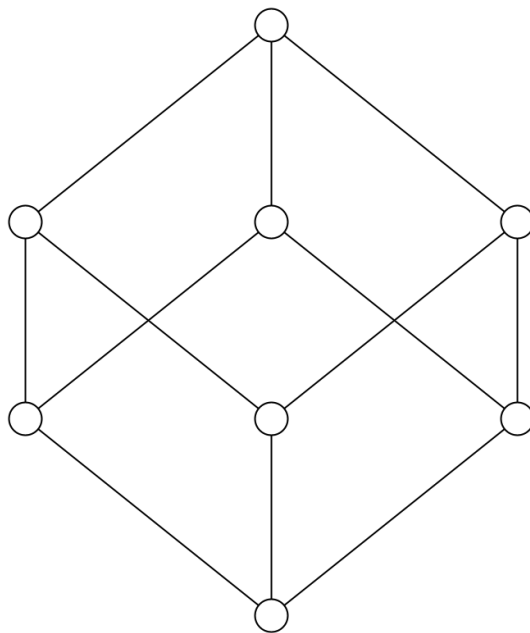


Dataflow Analysis in a Language Workbench

Master's Thesis



Matthijs Daniël Bijman

Dataflow Analysis in a Language Workbench

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Matthijs Daniël Bijman
born in Lochem, the Netherlands

© 2022 Matthijs Daniël Bijman.

Cover picture: Lattice structure, created using Processing

Dataflow Analysis in a Language Workbench

Author: Matthijs Daniël Bijman
Student id: 4490304
Email: m.d.bijman@student.tudelft.nl

Abstract

Dataflow analysis is a powerful tool used for program optimization, static analysis, and editor services for many programming languages. Spoofox, a language workbench, contains a domain-specific language called FlowSpec for the definition of control-flow and dataflow semantics that language developers can use to implement dataflow analyses for their language. FlowSpec however cannot be used to efficiently optimize programs. Other solutions are not suitable for language developers, or lack the ergonomics of a domain-specific language. In this thesis we present Flock: an incremental implementation of FlowSpec. We analyze the performance of Flock and show that it is efficient enough for use in optimization pipelines. Flock gives language developers the tools to succinctly write dataflow analyses for a wide variety of applications.

Thesis Committee:

Chair: Dr. ir. S. E. Verwer, Faculty EEMCS, TU Delft
Member: Dr. S. S. Chakraborty, Faculty EEMCS, TU Delft
University Supervisor: J. Smits MSc., Faculty EEMCS, TU Delft

Preface

This thesis contains the result of almost two years of work, concluding my seven years at the TU Delft. Even though the last two of these did not exactly go according to plan I am happy with the result in the form of this thesis. I am lucky to have worked on a topic that is incredibly interesting to me, and I hope I can put this knowledge to good use in the future.

There are some people that I need to thank for getting me through the this thesis project, as I am sure I would not have been able to complete it without them.

Foremost I need to thank Jeff Smits for his role as supervisor. It is a lot to ask someone to supervise a Master thesis, let alone one that takes two years to complete. I am lucky to have had a supervisor as calm and helpful as Jeff during this period.

I also need to thank my parents for their unending support, and for providing me with a space to go to whenever my room in Delft became too small.

And of course I want to thank all my friends and housemates who helped me stay sane and provided the necessary distraction. The weekly bouldering sessions with Tom were easily worth the tendonitis.

Finally I would like to thank Eelco Visser for sparking my interest in programming languages many years ago. His enthusiasm and mentorship during those years are what ultimately lead to me taking on this thesis project. I am sure he would have enjoyed reading the result.

Matthijs Daniël Bijman
Delft, the Netherlands
June 22, 2022

Contents

Preface	iii
Contents	v
List of Figures	vii
1 Introduction	1
1.1 Problem Statement	2
1.2 Research Questions	2
1.3 Contributions	2
1.4 Thesis Overview	3
2 Background	5
2.1 Program Optimization	5
2.2 Optimizations in Compilers	7
2.3 Compilers in a Language Workbench	11
3 Dataflow Analysis in a Language Workbench	15
3.1 Applications of Dataflow Analysis	15
3.2 Use cases and Limitations of FlowSpec	16
3.3 Other Implementations of Dataflow Analysis	17
4 Data Analysis in Flock	21
4.1 Control-Flow Specification	21
4.2 Data-Flow Specification	21
4.3 Queries, Optimizations, and Diagnostics	23
5 Solution Implementation	27
5.1 Flock Compiler	27
5.2 Flock Stratego API	28
5.3 Flock Runtime	30
5.4 Incremental Analysis in an Immutable Language	36
6 Evaluation	39
6.1 Run Time Analysis of an Optimization Pipeline	39
6.2 Memory Analysis of an Optimization Pipeline	42
6.3 Analysis of Query Latencies	43
6.4 Threats to Validity	43
7 Related work	47
7.1 Dataflow Analysis	47
7.2 Incremental Dataflow Analysis	47
7.3 Incremental Datalog	48
8 Conclusion	51
8.1 Future Work	52

Bibliography	55
A Tiger FlowSpec Control-Flow Spec	59
B Tiger FlowSpec Data-Flow Spec	63

List of Figures

2.1	A Tiger program that computes $10!$, the factorial of the value stored in x	7
2.2	Abstract Syntax Tree (AST) for the Tiger fact program	7
2.3	Simplified Control Flow Graph (CFG) for the Tiger fact program	8
2.4	Value analysis of the control-flow graph for the Tiger fact program in three steps. Analysis results are shown in blue. Analysed nodes are white, other nodes are gray.	8
2.5	A Tiger program that branches on a value and its corresponding control-flow graph	9
2.6	Worklist algorithm for solving forward dataflow analysis problems	10
2.7	Example of grammar specification with SDF3	11
2.8	Example of a for-loop matching the grammar rule	11
2.9	Example of the for-loop in ATerm form according to the grammar rule	11
2.10	A FlowSpec rule specifying the control-flow semantics of a For-loop node	11
2.11	A FlowSpec rule specifying the control-flow semantics of a VarDec node	12
2.12	A FlowSpec rule specifying the control-flow semantics of a FunDec node	12
2.13	FlowSpec dataflow rules that implement a liveness analysis	12
2.14	Rewrite rule to optimize a binary <i>and</i> -expression	12
2.15	Rewrite rule to apply optimizations to a program term	13
3.1	Program with value analysis results shown in brackets in the comments	16
3.2	Program with outdated analysis results after constant propagation	16
3.3	Program with updated analysis results after constant propagation	16
3.4	Program with available expressions between <code>{}</code> and liveness analysis between <code>[]</code>	17
3.5	Program with dead variable x eliminated, leaving invalid available expressions .	17
3.6	Program with unsound available expression optimization applied	17
3.7	A Stratego expression that creates a dynamic rule called <i>PropConst</i>	17
3.8	A strategy that defines generic forward dataflow analysis for an <code>if</code> term	18
4.1	Declaration of AST sorts and constructors in FlowSpec	22
4.2	Declaration of CFG rules in FlowSpec	22
4.3	Property definition for liveness analysis for Tiger in Flowspec	22
4.4	Dataflow rules for liveness analysis for Tiger in FlowSpec	22
4.5	Property definition and rules for value analysis for Tiger in Flowspec	23
4.6	Type definitions for value analysis for Tiger in FlowSpec	23
4.7	Property definition and rules for array length analysis for Tiger in Flowspec . . .	24
4.8	Generated Stratego API for liveness, value, and array length analysis	24
4.9	Optimization pipeline for Tiger in Stratego	24
4.10	Diagnostics that report out-of-bounds array accesses	25
4.11	Example of out-of-bounds diagnostics reported by array length analysis	25
4.12	Implementation of hover tooltips to show value analysis results	25
4.13	Example of on-demand dataflow analysis in an editor	26
5.1	Output file of the Flock compiler when compiling a FlowSpec Control Flow spec- ification	27
5.2	Output files of the Flock compiler when compiling a FlowSpec analysis	27
5.3	Pipeline stages of the Flock compiler	28

5.4	High-level overview of the runtime component structure	29
5.5	Steps performed during initialization	30
5.6	Steps performed after an analysis is queried	30
5.7	Steps performed after a transformation is perform	31
5.8	Replacement of term in Stratego term and Flock term. Note that P is unchanged in the Flock runtime, so that references into P remain valid.	31
5.9	Creation of CFG from a Flock term. Green nodes are entry nodes, red nodes are exit nodes.	32
5.10	Replacement of a (sub-)CFG	32
5.11	Resulting SCCs after a replacement with no cycles present.	33
5.12	Update to SCCs after a replacement within a cycle.	34
5.13	Adapted worklist algorithm used in the Flock runtime	35
5.14	Example of a dataflow rule that copies a map and changes a single entry	36
5.15	Class diagram of a compiled analysis.	36
6.1	Run time of optimization pipelines on various sizes of the program shown in Fig. 6.2	40
6.2	Example program for $n=3$	40
6.3	Run time of optimization pipelines on various sizes of the program shown in Fig. 6.4	40
6.4	Example program for $n=3$	40
6.5	Run time of optimization pipelines on various sizes of the program shown in Fig. 6.6	41
6.6	Example program for $n=3$	41
6.7	Run time of optimization pipelines on various sizes of the program shown in Fig. 6.8	41
6.8	Example program for $n=3$	41
6.9	Run time of optimization pipelines on various sizes of the program shown in Fig. 6.10	42
6.10	Example program for $n=3$	42
6.11	Normalization transformation that is applied before a peephole optimization can be used to fold the addition	42
6.12	Memory usage of Flock and Dynamic Rules in various benchmarks. Vertically the line represents the memory usage of the optimizer during the optimization of a program, horizontally the line represents a measurement as a percentage of total measurements. The graphs do not depict runtime.	43
6.13	Sorted latencies for value analysis queries shown for Flock (cached) and Dynamic Rules (uncached) on a logarithmic (left) and linear (right) scale	44
7.1	Control-flow graph rules for a C-style if in IncA (Szabó, Erdweg, and Voelter 2016)	49
7.2	Control-flow graph rules for a C-style if in FlowSpec	49
8.1	An optimizable program that does not match a peephole optimization	52
8.2	An optimizable program that will match a peephole optimization	52

Chapter 1

Introduction

Compilers are complex pieces of software that bridge the gap between code written by humans, and code understood by machines. Compilers were invented in the late 1940's and early 1950's, and reached widespread use with the development of the first high-level languages: Fortran, Cobol, and Lisp. By moving away from assemblers that required intimate knowledge of the hardware, programming was made easier and more robust. As resources were highly limited in the decades following their invention, the use of these languages remained constrained and somewhat controversial. For this reason, the developers of FORTAN aimed at generating efficient code that was better than handwritten assembler, to ensure that the language would be attractive to customers (Spencer 1997). Thus from the invention of high-level languages, optimization was an essential aspect of their implementation.

In stark contrast to the resource-limited machines from the 50's, the average person today owns several orders of magnitude more powerful hardware that fits in their pockets. Nevertheless, the demand for optimization in compilers has remained. The increase in computing power available comes with a seemingly even greater increase in computing power required. Moreover, the development of new programming languages with different semantics and constraints spurred the invention of new techniques to maintain sufficient performance. JavaScript for example, a dynamic scripting language invented in 1996 for programming websites, cannot be compiled ahead-of-time (AOT) because it is sent across the network to be run on unknown hardware. The widespread adoption of the language, new versions of the language, and the competition between implementations (called *engines*) keeps driving the complexity and effectiveness of optimizations forward to this day.

The design and implementation of compilers has evolved considerably since their invention in the 50's. Modern compilers include many steps, algorithms, and data structures, each responsible for a part of the translation process. At a high level, many compilers can be viewed as three stages. The *front-end* parses the textual input into an abstract-syntax tree (AST) format, performs language-specific analyses such as typechecking, and transforms the AST into an intermediate representation (IR). The *middle-end* performs optimizations on the IR that are independent of input language or target architecture. The *back-end* targets a single architecture, performing architecture-specific optimizations, and transforming the IR into assembly. This modular three-tiered architecture has several advantages over more rigid architectures. Implementing a new language requires only a new-frontend, while the middle-end and back-ends can be reused. Furthermore, new optimizations in the middle-end or newly supported architectures in the back-end can be used by many different languages. A prominent example of this architecture can be found in LLVM (C. A. Lattner 2002; C. Lattner and Adve 2004), which contains the language agnostic middle- and back-ends. New languages can be implemented by translating to the LLVM IR, and the rest of the compilation process is performed by LLVM.

Many optimizations used in compilers such as LLVM are based on a technique called *dataflow analysis*. Using this technique, we can safely approximate the state of a program at various points during its execution. By choosing the various parameters of dataflow analysis we can approximate many different types of behaviour. We can analyse the values computed during program execution, which variables are never used, expressions that are redundantly recomputed, and many more. These analyses can then be used to implement optimizations.

1.1 Problem Statement

The three-stage architecture by projects such as LLVM allows language developers to harness the power of dozens or hundreds of optimizations without additional effort beyond the translation from their source language to LLVM bytecode. While a great benefit, LLVM may not be a suitable target for every language. A language can be highly dynamic, interpreted, or otherwise embedded in a wider runtime and thus may not be suitably compiled by LLVM. Furthermore, a language may require or benefit from specific types of analyses that are not present in LLVM. The analyses present in LLVM can also not be used for purposes besides optimization, such as advanced diagnostics (*Jetbrains Dataflow Analysis* 2022). Due to one or more of these constraints, a language developer may need to manually implement the necessary dataflow analyses. This however reintroduces the challenges that LLVM alleviates, including efficient and correct implementation of the analyses and optimizations, and phase-ordering of analyses and optimizations.

In this thesis we investigate these challenges in the context of the language workbench Spoofox. Spoofox includes several metalanguages for the implementation of programming languages, and more specifically DSLs. Since LLVM bytecode is rarely a suitable target for DSLs developed with Spoofox we see potential for a valuable addition to the toolset of the workbench. Furthermore we find that while research exists in area of composition of analyses and optimizations, this work does not exist in the context of language workbenches, where flexibility, usability, and integration with the larger language implementation are important aspects.

1.2 Research Questions

The research questions that we aim to answer based on the problem statement are as follows:

- **RQ1** What does a flexible but efficient dataflow analysis and optimization framework for language developers look like within Spoofox?
- **RQ2** How can we efficiently and automatically compose dataflow analyses and optimizations without imposing strong restrictions on language developers?
- **RQ3** How does the efficiency of such a framework compare to other approaches when applied to optimization and analysis?
- **RQ4** How much effort is required to implement an optimization pipeline, static analysis, or editor services with such a framework?

1.3 Contributions

This thesis contains contributions to the problem of using dataflow analysis for the implementation of languages developed within a language workbench. It contains a compiler and runtime for FlowSpec, a DSL for dataflow analyses, that generates incremental implementations of analyses. The runtime exposes a Stratego API that can be used to query analyses and perform program transformations. To evaluate the framework we use quantitative benchmarks to understand the performance of the framework compared to other optimization frameworks. We investigate the flexibility by implementing both optimizations, static analysis, and editor services with the framework. The key contributions of this thesis can be summarized as follows:

- A method for incrementally updating control-flow graphs and their strongly connected components following program updates given a high-level control-flow specification
- A method for deriving incremental dataflow analyses from a high-level dataflow specification
- An implementation of the FlowSpec language that can be used within the Spoofox language workbench to implement optimization pipelines, static analyses, and editor services

1.4 Thesis Overview

The remainder of this thesis consists of seven chapters. In Chapter 2 we introduce and explain the concepts that underlie this thesis to provide the necessary background information. This chapter can be (partially) skipped if the reader is familiar to the material. In Chapter 3 we investigate the problem presented here in more detail. In Chapter 4 we show our approach to solving this problem, followed by a discussion of the implementation in Chapter 5. Chapter 6 contains the evaluation of the solution, containing benchmarks of a variety of analyses and optimizations applied to the Tiger language. Chapter 7 contains a discussion of related work. Finally, in Chapter 8 we summarize the thesis and discuss future work.

Chapter 2

Background

In this chapter we will take a dive into the problems and techniques used in program optimization. The goal is to provide the background knowledge necessary to understand both the problem definition and the proposed solution of this thesis. This assumes some basic level of familiarity with programming, and algorithms used in compilers.

The field of program optimization is incredibly broad, and as such there are many interesting subjects that we cannot discuss here. Optimization happens during many stages in a compiler with a wide variety of techniques, each with their own strengths and weaknesses. We will not discuss all of these techniques, instead we focus on the material necessary to understand the work in this thesis.

This work in this thesis performs optimizations using *dataflow analysis*, a technique widely used for optimization. We will begin by discussing some examples of optimizations that can be performed using dataflow analysis in Section 2.1. In Section 2.2 we look at the various components that make up an optimizing compiler, from parsing programs to analysing and transforming them.

2.1 Program Optimization

In this section we will take a first look at optimizations. We will present some programs in a language called Tiger (Appel 2004), and consider possible optimizations that may be applied to the programs to make them more efficient. Tiger is a simple language (relative to mainstream languages such as C++ and Java), but it contains many features that are interesting to study and translate well to real-world cases. This includes features such as functions, loops, IO, and arithmetic and boolean operations.

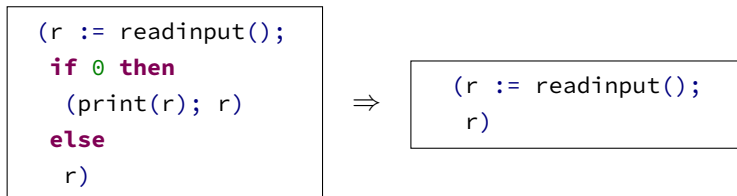
Propagating and Folding Constant Values

We will start our tour of dataflow-based optimizations by looking at an optimization called *Constant Propagation*. Constant propagation replaces variables with their corresponding *constant* value if possible. The resulting program the often be further optimized by performing *Constant Folding*. Constant folding transforms expressions such as $1 + 1$ or $3 * 3 * 3$ into their results, 2 and 27 respectively. We can compute such expressions once during compile time to avoid computing them (possibly repeatedly) during run time. An example of a program optimized in this manner is shown here:

<pre>(a := 1; b := a + 1; c := b + a)</pre>	\Rightarrow	<pre>(a := 1; b := 2; c := 3)</pre>
--	---------------	--

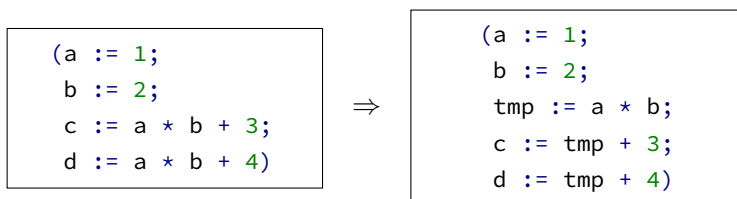
Dead Code Elimination

After applying other optimizations it is common to uncover paths in the code that can never be executed. For example, a piece of code that conditionally logs some information based on a debug flag will never run if the debug flag is disabled. After propagating the constant to the branch, we can perform dead code elimination:



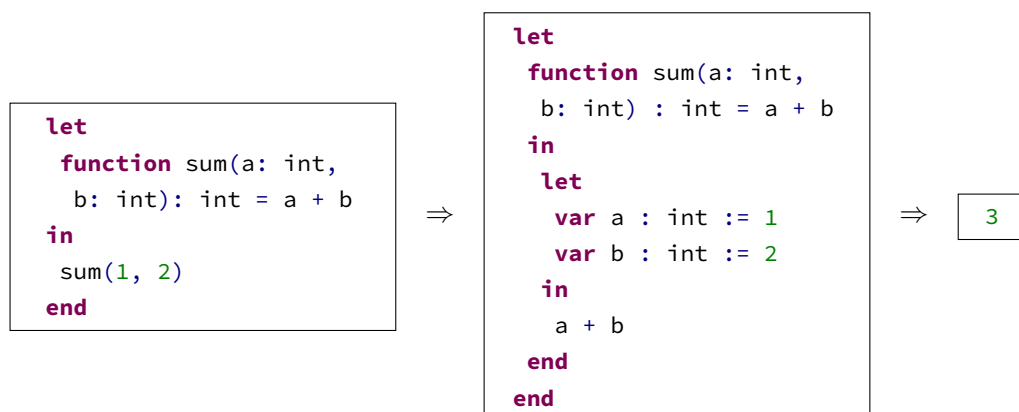
Common Subexpression Elimination

Common subexpression elimination is an optimization that removes redundant computation by identifying expressions that are computed multiple times with identical results. The following example shows a program with common subexpressions that are optimized by introducing a new variable that stores the common expression:



Function Inlining

The final optimization we will discuss is not quite an optimization. *Function Inlining* is the transformation that replaces a function call with the body of the corresponding function. While this will remove the overhead of performing a function call, it can also degrade performance by increasing instruction cache pressure, program size, etc. Regardless, function inlining is an important transformation because it unlocks other optimizations. In this example we can see how function inlining makes it possible to perform additional constant propagation and folding optimizations:



```

let
  function nfactor(n: int): int =
    if n = 0 then 1 else (n * nfactor(n-1))

  var x: int := 10
in
  nfactor(x)
end

```

Figure 2.1: A Tiger program that computes $10!$, the factorial of the value stored in x

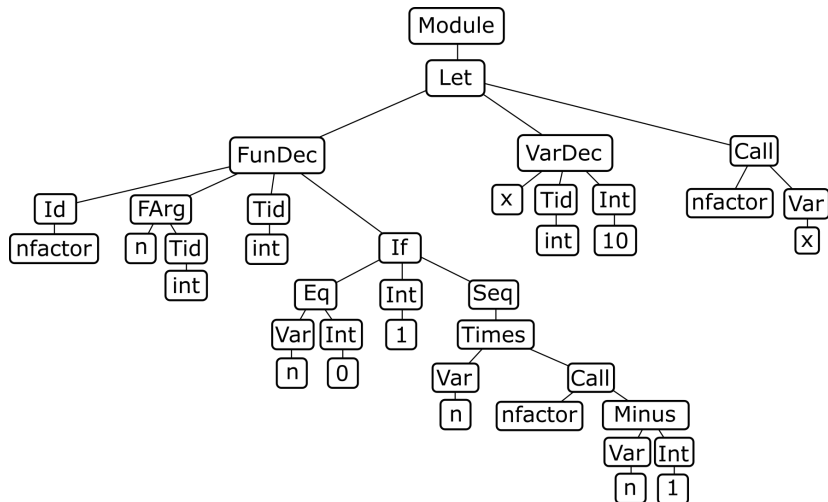


Figure 2.2: Abstract Syntax Tree (AST) for the Tiger fact program

2.2 Optimizations in Compilers

In this section we will discuss how the optimizations we introduced in the previous section can be implemented in a compiler. We will begin by describing how code is parsed into various structures that the compiler can process efficiently. Then we describe how the compiler can analyse these structures to discover and apply optimizations. As a running example we will take the Tiger program shown in Fig. 2.1. It computes the factorial of whatever value is stored in x .

The Abstract Syntax Tree

One of the first steps a compiler takes when it compiles a program is to turn the textual input written by the programmer into a structure that is easier to work with. The textual input consists of one or more pieces of text which the compiler transforms into an *abstract syntax tree* (AST). This process is called *parsing*. The rules that decide what the AST of a program looks like is called the *grammar*. The result of parsing is a tree datastructure that can be transformed, searched, iterated, etc. more efficiently than in text form due to the explicit relations between parts of the program. This datastructure forms the basis of all other steps of a compiler. When a compiler performs optimizations they are applied on the AST. The optimized AST is then used by later stages, such as translation to binary code.

The AST also lends itself well to visual representation. Fig. 2.2 shows the AST corresponding to the program shown in Fig. 2.1. We can see some important characteristics of ASTs in this visualisation. An AST has a single root node, in this case the *Module*, representing the entire module (or file). Furthermore, in contrast to textual input the AST clearly subdivides parts of the program through child/parent relations. This is a key advantage of the AST, as the compiler can efficiently lookup elements of the program by traversing the tree. Finally, we see that the *values* in our Tiger program are present as *leaf nodes*.

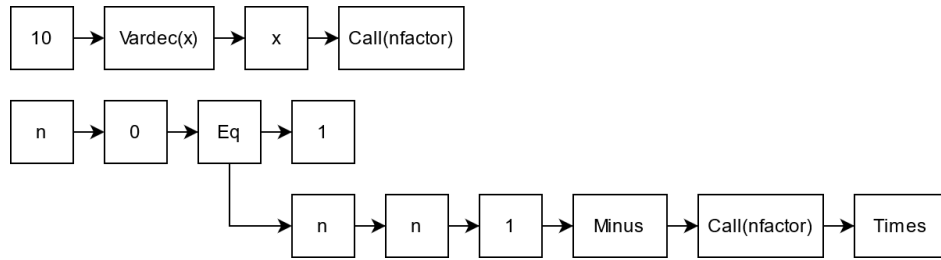


Figure 2.3: Simplified Control Flow Graph (CFG) for the Tiger fact program

The Control Flow Graph

The AST is easier to manipulate for a compiler, but there are more suitable and useful representations when optimizing programs. One such representation is a control flow graph (CFG). The CFG for the factorial program can be seen in Fig. 2.3. The CFG is a derivative of the AST in which control-flow is explicit in the graph. In the graph, nodes correspond to expressions in the program, while the possible transitions between expressions are (directed) edges. This means that when we are traversing the CFG through its edges, we are traversing the nodes of the program in the same way they are traversed when it is executed. Consequently when we want to analyse our program in a way that depends on the control-flow, we do not have to look at the types of AST nodes we are traversing, and instead can look at the edges of our CFG.

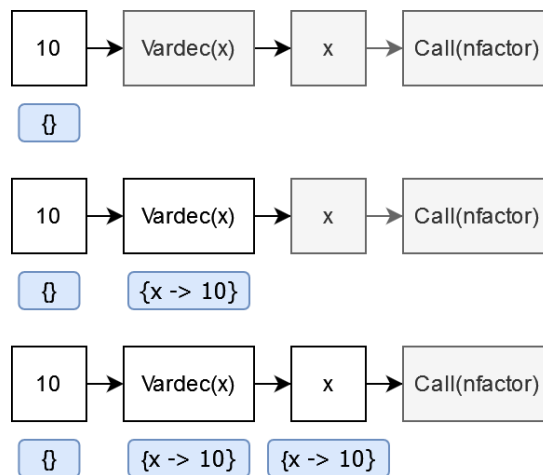


Figure 2.4: Value analysis of the control-flow graph for the Tiger fact program in three steps. Analysis results are shown in blue. Analysed nodes are white, other nodes are gray.

Dataflow Analysis

Once the control-flow graph is constructed, we can perform program analysis to discover optimizations. In this section we will describe how we can use a control-flow graph to analyze a program with the aim of finding optimizations that can be applied to it. A common type of program analysis is called *dataflow analysis*, and it is the main tool used in this thesis to analyze and optimize programs. We have seen examples of optimizations that we can perform using dataflow analysis in 2.1.

Dataflow analysis can be used to approximate the behaviour of a program without executing it by simulating the behaviour of the program nodewise while traversing the control-flow graph. Using the analysis results the compiler can determine if an optimization can be safely applied without changing the semantics of the program.

A dataflow analysis can be used to analyse which values are stored in variables at each point in the program. For example, consider Fig. 2.4 which shows in three steps how analysis results propagate from a *VarDec* node to a variable reference node.

Other instances of dataflow analysis may record different information about the program (such as computed expressions, or read variables), or propagate in the reverse direction over the control-flow graph (as is the case when we compute which variables are live).

Dataflow analysis becomes more complex when we introduce branches and loops in the control-flow graph. As we are trying to find an approximation of the behaviour of the program, we must decide how information is merged when two paths come together. Consider the program and its control-flow graph shown in Fig. 2.5.

When we perform the same value analysis on this new control-flow graph, we find that the value of y at the last node of the graph depends on the path taken. We can easily tell that the program will always assign 2 to y , but our analysis cannot. Furthermore, we can encounter programs where we can never be sure what path is taken. Imagine for instance a program that prints the current day of the week. We cannot know in advance on which day of the week the program is executed.

The solution to this problem is to choose an *abstract domain* to represent the values of our program, and define operators that merge multiple values of this domain together. The choice abstract domain is generally a tradeoff between precision and performance. For instance, we can choose the abstract domain of a single value. In this case merging two different values is not possible, and the analysis will propagate a \top value, indicating a lack of precision. Another option is to choose the abstract domain of the powerset of values. In this case we can merge values by performing a set union. The analysis result will then be $\{2, 3\}$, containing the possible values for y . A drawback of this approach is that the set of values might grow very large, impacting the performance of the analysis.

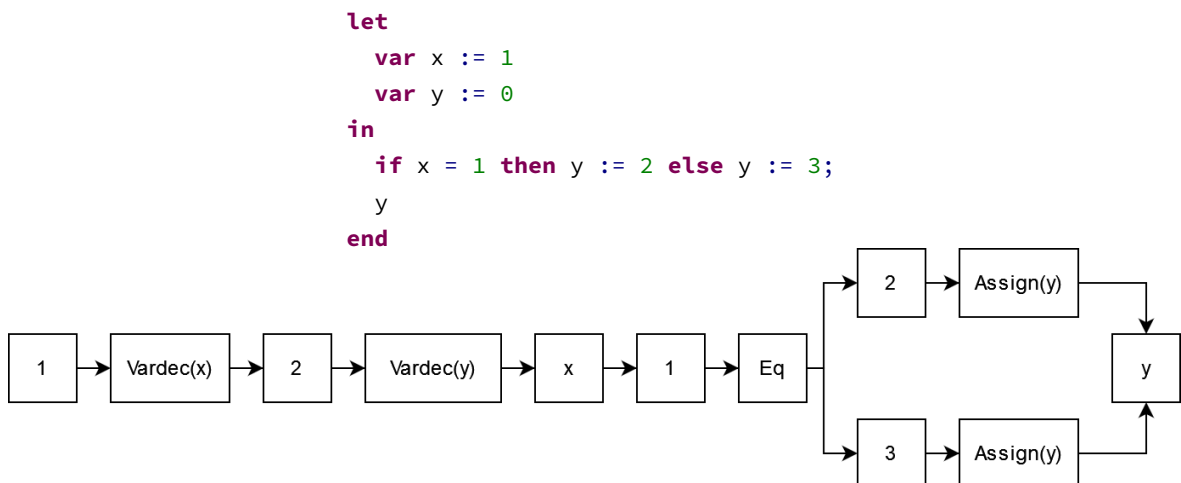


Figure 2.5: A Tiger program that branches on a value and its corresponding control-flow graph

Formal Definition

In this section we will formally introduce dataflow analysis. We will use the precise definition from Marlowe and Ryder (1989). A dataflow analysis consists of a tuple (G, L, F, M, n) where:

- G is a rooted digraph of (V, E, p) , vertices, edges, and the root respectively
- L a semilattice
- F a space of functions mapping L into L
- M a mapping of graph edges E into F
- n is an element of L

The rooted digraph G is the control-flow graph representation of a program or procedure, where the root represents its entry point. In practice we create a separate rooted digraph for each procedure in the program. A semilattice is the mathematical object that we use to approximate information about our program. It represents the abstract domain of our analysis. A (semi)lattice *instance* represents the analysis results computed at its corresponding control-flow node. The flow function (an element of F) is associated with graph edges through mappings in M . It determines how information (represented as an instance of the chosen semilattice) is propagated along that edge. n is the initial value in L associated with the root p of the graph.

We also define a solution S to the dataflow problem as a mapping from V to L , representing the semilattice instance computed at each program point. The result of our analysis S maps V to L , such that we have a lattice value for each vertex.

We obtain a fixpoint solution to our dataflow problem when evaluating any of the flow functions corresponding to the edges of the graph does not change S .

Performing Dataflow Analysis

Given the precise definition above, we would like an algorithm to compute the solution S for our program, such that the compiler can implement and use the dataflow analyses. A simple method and common method for solving a dataflow problem is using an iterative worklist algorithm, shown in Fig. 2.6.

The algorithm first adds all vertices of the graph to the worklist, to ensure that each vertex is processed at least once. The algorithm then iteratively takes an element v of the worklist, applies the flow functions from to each of its incoming edges, joins the results, and updates the solution corresponding to v in S . If the solution is changed, the successors of v are added to the worklist. For a backwards dataflow problem (such as liveness analysis) we instead add the predecessors.

When none of the flow functions change the results in S , the work list will be empty, thus we have reached a fixpoint.

```
worklist = {}
for v in V:
    initialize v
    add v to worklist

while len(worklist) > 0:
    v = worklist.pop
    recompute the solution at v
    if the solution changed:
        add successors to worklist
```

Figure 2.6: Worklist algorithm for solving forward dataflow analysis problems

```
Exp.For = <
  for <Var> := <Exp> to <Exp> do
    <Exp>
  >
```

Figure 2.7: Example of grammar specification with SDF3

```
for a := 1 to 5 do
  print("in a loop")
```

Figure 2.8: Example of a for-loop matching the grammar rule

```
Mod(
  For(
    Var("a")
    , Int("1")
    , Int("5")
    , Call("print", [
      String(
        "in a loop"
      )
    ])
  )
)
```

Figure 2.9: Example of the for-loop in ATerm form according to the grammar rule

2.3 Compilers in a Language Workbench

In this section we will discuss how the various techniques from the previous section are implemented in a compiler in the context of a language workbench, Spoofox (Kats and Visser 2010). Spoofox contains a collection of declarative metalanguages (languages for developing languages) that aim to reduce the implementation effort for language developers.

2.3.1 Parsing with SDF3

Syntax Definition Formalism 3 (SDF3) (Souza Amorim and Visser 2020) is a metalanguage that automates the implementation of the parsing algorithm given a language grammar. It allows the compiler developer to specify the grammar rules of a language declaratively, and receive a program that will parse program written in their language into an AST, in ATerm format. Fig. 2.7 shows an example of a grammar rule for Tiger for-loops in SDF3, with a snippet of Tiger code that follows the rule shown in Fig. 2.8. Finally Fig. 2.9 shows the resulting ATerm from parsing the snippet according to the grammar rule. Besides these grammar rules, we can also specify syntax elements such as operator associativity, precedence, formatting, keywords, and more. We use a syntax definition written in SDF3 for our Tiger compiler.

2.3.2 Control-Flow Graphs with FlowSpec

FlowSpec (Smits and Visser 2017) is a metalanguage, similar to SDF3, for specifying control-flow graph semantics of a language. It aids a compiler developer in implementing the process of converting an AST into a CFG. FlowSpec will be discussed in more detail in Chapter 4 because the work in this thesis includes an implementation of FlowSpec, however we will discuss how we can use FlowSpec to construct CFGs from ASTs here.

```
For(binding, body) =
  entry -> binding -> body -> exit,
  body -> binding
```

Figure 2.10: A FlowSpec rule specifying the control-flow semantics of a For-loop node

Consider a FlowSpec rule that builds the CFG of a for loop in Tiger in Fig. 2.10. The control-flow starts at the loop binding, and then either traverses the body and returns to the binding, or exits the loop. FlowSpec will create the sub-CFGs for the binding and body, and create edges between them as the FlowSpec declaration specifies. The *entry* and *exit* names specify how the edges between the parent CFG and this sub-CFG should be created.

When executed, the rule in Fig. 2.10 creates edges between sub-CFGs, but it does not create CFG nodes itself. An example of how we can create CFG nodes is shown in Fig. 2.11.

```
VarDec(n, t, e) = entry -> e -> this -> exit
```

Figure 2.11: A FlowSpec rule specifying the control-flow semantics of a `VarDec` node

When declaring a variable in Tiger, the expression will be evaluated before the declaration itself is evaluated. This mirrors how the program stores the result of the expression in the variable. To show this in a CFG we can use the *this* name in FlowSpec rules. When used, FlowSpec creates a node corresponding to the outermost AST node that was pattern matched in the rule.

```
root FunDec(n, args, rt, body) = start -> body -> end
```

Figure 2.12: A FlowSpec rule specifying the control-flow semantics of a `FunDec` node

Finally, we need to be able to specify which AST nodes form the root of our CFG. Shown in Fig. 2.12 is a FlowSpec rule that specifies that every function declaration is a root, and thus requires its own CFG. This implies that a Tiger program with multiple function declarations will have multiple CFGs. Instead of *entry* and *exit*, we use *start* and *end*, since they correspond to the start and end of the entire CFG.

2.3.3 Dataflow Analysis with FlowSpec

Aside from creating control-flow graphs, FlowSpec also allows you to specify dataflow rules to implement analyses. Fig. 2.13 contains an example of an implementation of liveness analysis. We will informally describe how this analysis maps to the formal definition from Section 2.2.

The input of our FlowSpec analysis is a CFG created from the control-flow graph rules, matching graph G . The semilattice L is the result type of our analysis rules, which is a set of variable names. Sets are part of FlowSpec, and implement the required lattice operations. The flow functions F are the bodies of the dataflow rules, while the pattern matches on the lhs of the rules provide the mapping M between edges and functions (the rhs). Finally, the first dataflow rule specifies the initial/default lattice value, the empty set, corresponding to n .

Given this mapping (which we can extend to include more FlowSpec features) we can treat any FlowSpec analysis as a dataflow analysis, and solve it using the worklist algorithm shown above.

```
live(_.end) = {}
live(VarDec(n, _, _) -> next) = {m | m <- live(next), m != n}
live(Var(n) -> next) = {n} \/ live(next)
live(_ -> next) = live(next)
```

Figure 2.13: FlowSpec dataflow rules that implement a liveness analysis

Transformations with Stratego

In this section we will describe how we can perform transformations on our program. Unlike previous sections where we described the theory and tools separately, the method used to implement transformations is highly dependent on the language and datastructures used to implement a programming language. Spoofox however contains a metalanguage for transforming programs, called Stratego (Visser 2004). Stratego allows a language developer to describe transformations on program terms and compose them together to implement systems such as compilers.

```
opt: And(Var(a), Var(a)) -> Var(a)
```

Figure 2.14: Rewrite rule to optimize a binary *and*-expression

In the context of optimizers, Stratego allows us to traverse our AST and look for optimization opportunities by inspecting the structure of terms. As an example, consider the

rule defined in Fig. 2.14. It defines a simple optimization called *opt* that turns an expression of the form $n \ \& \ n$ into n . The rule cannot be used on its own however; we need a rule that takes a program term, and tries to find subterms where it can apply the *opt* rule.

```
opt-program: Mod(terms) -> Mod(<topdown(try(opt))> terms)
```

Figure 2.15: Rewrite rule to apply optimizations to a program term

Fig. 2.15 shows a rule that performs the desired transformation. It will try to apply our *opt* rule to each subterm of our program, in topdown manner (from root to leaf terms). When we execute this rule on a program with an expression like $n \ \& \ n$, it will produce a new program with the optimization applied.

While this approach is well suited for many types of program transformations, it is not a good fit for complex optimizations that require dataflow analysis. For this reason Stratego was extended to include support for dynamic rules (Bravenboer et al. 2006). Dynamic rules make it possible to simulate the effects of terms while traversing the program, similar to dataflow analysis. We discuss the use of dynamic rules for dataflow analysis in more detail in Ch. 3 and 6.

Correctness in Language Development

As we have seen so far the goal of optimization is to produce a faster program. Of course we would also like the optimized program to behave similarly to our input program. This constraint on our optimizer is called *correctness*. The definition of correctness depends on the language that we are optimizing.

For Tiger, we want our optimized programs to produce the same output (the accumulation of all printed strings) as our unoptimized programs. If we only have the output of a program to go by, we should not be able to determine whether it is optimized or not. For Tiger this means that memory usage and run time can be changed by the optimizer freely. In other domains such as cryptography constraints can be much stronger. Compilers may inadvertently introduce timing vulnerabilities to cryptographic libraries (Pornin n.d.). Correctness thus not only depends on the language, but also the domain of the program.

For more complex languages such as *Java*, *C++*, *etc.* there are many more ways the language interacts with its environment. A programmer can edit files on the filesystem, send data over the network, play sounds and video, etc. Each of these actions may need to be conserved by the compiler for the program to remain *semantically* identical.

Determining if a compiler is correct is a very difficult problem. Formal proofs fall most commonly in one of two methods. Either we prove that the compiler is correct for all possible inputs, or we prove (automatically) that the compiler has correctly compiled a particular input. Proving a compiler to be correct is an exceedingly difficult problem, and examples of it are rare. CompCert (Leroy et al. 2016) is a prominent example of a verified C compiler.

More commonly compilers are not formally verified, and instead thoroughly *tested* with a variety of approaches. Unit tests, regression tests, and fuzzing are common techniques used to improve robustness. Even though mature compilers are tested to an extreme extent because of the potential consequences of a bug, there is no guarantee of correctness like a verified compiler can give. Yang et al. (2011) show that even mainstream compilers with millions of users like GCC and LLVM contain a myriad of bugs. The possibility remains that a fault in the compiler slipped by each test case, to be discovered by an unlucky programmer.

In this thesis we do not consider correctness as part of our research goals, as it is outside our scope.

Chapter 3

Dataflow Analysis in a Language Workbench

In this chapter we will investigate the challenges involved in building an efficient and flexible dataflow analysis framework that fits within the Spoofox environment. We begin by setting requirements based on a variety of use cases, so that we can evaluate existing solutions. We then consider the capabilities of the existing FlowSpec implementation, and why these are not sufficient for implementing an optimization pipeline. Then we investigate two other approaches that use dataflow analysis and evaluate them against our requirements.

3.1 Applications of Dataflow Analysis

In this section we discuss the three applications that we wish to support with a framework for dataflow analysis. We base these requirements on the use cases for dataflow analysis that we find in literature and industry.

3.1.1 Optimization

The first and foremost application of dataflow analysis is in optimization by compilers. As we have seen in Ch. 2, dataflow analysis forms the basis for a variety of optimizations. It is widely used by industrial compilers, such as GCC (Hayes 1999) and LLVM (C. Lattner and Adve 2004).

3.1.2 Program Checking

Besides optimization, a common use for dataflow analysis is in program checking. A prominent example is Java, in which a field marked `final` must be initialized in the constructor of a class (Gosling et al. 2000). This check is performed using dataflow analysis. Similarly the Rust language implements a variety of checks including finding uninitialized variables, determining borrows at each program point, and determining live variables across `yield` statements, using dataflow analysis (*Dataflow Analysis* 2022). The Clang frontend similarly uses dataflow analysis to detect bugs (*Clang Dataflow* 2022).

3.1.3 Diagnostics

A variant of program checking is the use of dataflow analysis for diagnostics. Diagnostics in the form of IDE warnings do not prevent a user from compiling a program, but they provide feedback on potential errors or inefficient operations. Analysis can also be initiated by the user (*Jetbrains Dataflow Analysis* 2022), to aid in understanding or debugging the code.

3.2 Use cases and Limitations of FlowSpec

FlowSpec (Smits and Visser 2017) is a declarative domain specific language for defining control-flow and data-flow rules. It allows a language developer to succinctly define dataflow analyses over a Spoofax-based language. The implementation of a Read-Write analysis for the Green-Marl (Hong et al. 2012) language has shown its real-world applicability to an extent, however there are other use cases which FlowSpec cannot efficiently support. Crucially the evaluation in Smits and Visser (2017) is limited to analyses that are executed once, on the entire control-flow graph, due to limitations in the implementation of FlowSpec. This constraint prevents its use in other common use cases, the most important of which is optimization pipelines. In this section we will investigate why this is the case.

Optimizations Invalidate Analyses

Running a dataflow analysis to a fixpoint before looking for optimizations is inefficient due to the interaction between optimizations and analyses. Because dataflow information propagates through the edges of the control-flow graph, a change to node n may change the analysis results at all nodes reachable from n in the CFG. Consequently if we wish to look for further optimizations in the program we must perform the analysis again, updating the outdated information.

As an example consider the program in Fig. 3.1. The analysis results computed initially tell us that the value of y is unknown when it is printed (indicated by the *top* value). However we can perform an optimization by propagating the constant assigned to x to the assignment to y , shown in Fig. 3.2. After this transformation, our analysis will discover that y has a constant value at the time of printing. The old analysis results are outdated, and the program must be reanalysed to discover the new optimization.

```

let
  var x: int := 10
  // {x -> 10}
  var y: int := x
  // {x -> 10,
  // y -> top}
in
  print(y)
  // {x -> 10,
  // y -> top}
end

```

Figure 3.1: Program with value analysis results shown in brackets in the comments

```

let
  var x: int := 10
  // {x -> 10}
  var y: int := 10
  // {x -> 10,
  // y -> top}
in
  print(y)
  // {x -> 10,
  // y -> top}
end

```

Figure 3.2: Program with outdated analysis results after constant propagation

```

let
  var x: int := 10
  // {x -> 10}
  var y: int := 10
  // {x -> 10,
  // y -> 10}
in
  print(y)
  // {x -> 10,
  // y -> 10}
end

```

Figure 3.3: Program with updated analysis results after constant propagation

Optimizations Invalidate Optimizations

While imprecise, the outdated analysis results shown in Fig. 3.2 are not incorrect. This means we can continue looking for optimizations even though the information is not up-to-date. It may seem that we can exhaustively apply optimizations before reanalysis, but it is also possible that the analysis results become *unsound* after an optimization is performed. Consider the program in Fig. 3.4, on which both liveness and available expressions are analysed. Liveness analysis tells us the assignment to x is dead and can be removed, however this leaves unsound available analysis information. The assignment y is replaced with x , because the analysis result tells us x also holds $2 * 2$.

Whereas in the previous case, we were able to soundly continue looking for optimizations after the analysis results became outdated, we can see it is also possible for analysis results

```

let
  var x: int := 2 * 2
    // {x -> 2 * 2}
    // []
  var y: int := 2 * 2
    // {x -> 2 * 2,
    // y -> 2 * 2}
    // [y]
in
  print(y)
    // {x -> 2 * 2,
    // y -> 2 * 2}
    // [y]
end

```

Figure 3.4: Program with available expressions between {} and liveness analysis between []

```

let
  var y: int := 2 * 2
    // {x -> 2 * 2,
    // y -> 2 * 2}
    // [y]
in
  print(y)
    // {x -> 2 * 2,
    // y -> 2 * 2}
    // [y]
end

```

Figure 3.5: Program with dead variable x eliminated, leaving invalid available expressions

```

let
  var y: int := x
    // {x -> 2 * 2,
    // y -> 2 * 2}
    // [y]
in
  print(y)
    // {x -> 2 * 2,
    // y -> 2 * 2}
    // [y]
end

```

Figure 3.6: Program with unsound available expression optimization applied

to become unsound, requiring a reanalysis before we can continue optimizing. Due to these interactions, using FlowSpec for optimization pipelines as-is is highly inefficient.

3.3 Other Implementations of Dataflow Analysis

In this section we will look at two other approaches to dataflow analysis. First we will look at another approach in Spoofox to data-flow analysis and optimization from Bravenboer et al. (2006), followed by a look at dataflow analysis in LLVM (C. Lattner and Adve 2004). In Chapter 6 we compare the performance and flexibility of both approaches against our own.

Dynamic Rules in Spoofox

Dataflow analysis and optimization can be implemented using dynamic rules in Spoofox (Bravenboer et al. 2006). We will briefly introduce dynamic rules and look at an example of an optimization, constant propagation, implemented using them.

Dynamic rules are rewrite rules that can be dynamically created and can capture variables from the context in which they are created. Fig. 3.7 shows how we can create a dynamic rule that maps a *Var* term to its assigned value. After running the rule on an appropriate *Assign* term with a constant rhs, we can invoke the *PropConst* rule on an identical *Var* term and receive the constant rhs. While traversing the program (in the order of execution) we can create dynamic rules whenever we encounter a constant assignment, and invoke them whenever we encounter a variable reference. If this invocation succeeds, it means the variable holds a constant value and we can perform the optimization.

```

prop-const-assign =
  ?Assign(x, e)
  ; where(<is-value> e)
  ; rules(PropConst : Var(x) -> e)

```

Figure 3.7: A Stratego expression that creates a dynamic rule called *PropConst*

Dynamic rules in Spoofox support some additional features that are necessary to fully implement dataflow analysis, which we will briefly mention here. First is overwriting/undefining rules, which can be used to undefine a rule with respect to a term, to make sure that previously created dynamic rules cannot be invoked anymore. Second are rule scopes,

to limit the lifetimes of rules, similar to how variables can be scoped in programs. Finally, it is possible to perform a union or intersection over rule sets (the set of rules with the same name) to implement the equivalent of lattice glb or lub operations, which are necessary to perform analysis over branches. For loops there is support for fixpoint iteration.

The dynamic rule approach to dataflow analysis has been shown to be a very effective tool for implementing optimizers (Bravenboer et al. 2006). We believe this same approach can support both program checking and simple diagnostics, as they follow a similar pattern: traverse the program while performing analysis, albeit without transformations.

One drawback of dynamic rules is the lack of caching, which impacts the latency when using dynamic rules-based dataflow analysis as an editor service. A more significant drawback is that Dynamic Rules is a relatively low-level operational encoding of dataflow analysis. Fig. 3.8 shows an example of dataflow analysis with Dynamic Rules for an `if` term. The snippet shows several drawbacks of such an operational encoding. Control-flow semantics are implicitly encoded by the order of `recur` applications, the merging of dataflow information must be explicitly specified, and optimizations must be performed in lock-step with the traversal of the program. Ideally we can decouple the optimization from the control-flow semantics, and automatically merge dataflow information where necessary.

We include dynamic rules in our benchmarks in Ch. 6.

```
forward-prop-if(transform, before, recur, after | Rs1 , Rs2 ) =
    ?If(b, t, e)
    ; !If(<recur> b, <id> t, <id> e)
    ; (transform
    <+ before
    ; (?If(b, t, e); !If(<id> b, <recur> t, <id> e)
    /~Rs1 \~Rs2 / ?If(b, t, e); !If(<id> b, <id> t, <recur> e))
    ; after)
```

Figure 3.8: A strategy that defines generic forward dataflow analysis for an `if` term

LLVM

While the scope of LLVM far exceeds dataflow analysis and optimization, it is an important part of the project. LLVM provides language developers the power of dataflow analysis by providing a compilation target called LLVM IR. A language developer must provide a frontend that can translate the source language into LLVM IR, and LLVM will take care of optimization and code generation. This approach has proven to be very successful, and there are popular frontends for a wide variety of languages including C, C++, Rust, Fortran, Go, etc.

The optimizer and code generation capabilities of LLVM are state-of-the-art, and reproducing those capabilities is far outside the scope of most language developers. There are however some downsides to this approach. First of all, the translation from source language to LLVM IR can mean losing semantic information about the program. Consider the type system of a functional language such as Haskell. It provides strong guarantees on the behaviour of the program due to characteristics such as laziness and purity, which enables a Haskell compiler to apply optimizations that LLVM cannot.

A second disadvantage is that compiling to LLVM IR locks you into the existing LLVM backends. Many languages, especially DSLs, cannot easily be compiled to binary format as they are designed to run within a runtime that is not compatible with a binary artifact. LLVM does include a JIT engine, which makes it a more widely usable target for dynamic languages, but this has not seen the level of adoption the AOT compiler has. For instance, the WebKit project replaced their LLVM based JavaScript JIT with a custom implementation (Pizlo 2016).

The conclusion is that LLVM is an excellent for optimization using dataflow analyses, if none of the drawbacks apply. The foremost reason it may not suitable is if the execution

targets supported by LLVM are not sufficient. It is also not a solution for languages that require dataflow analyses for program checking, nor can it aid in the implementation of (interactive) diagnostics (although LLVM can still support other parts of the compilation process). We also include LLVM in our benchmarks in Ch. 6.

Chapter 4

Data Analysis in Flock

Flock is our implementation of FlowSpec with support for incremental analysis. It integrates into the Spoofox language workbench to provide language developers with the tools to write and execute control-flow and dataflow specifications. To validate our implementation we have implemented an optimizer and editor services for the Tiger language. This chapter serves as an exposition of the FlowSpec language and Flock runtime. We discuss both the FlowSpec specifications, as well as the Stratego code that interfaces with the Flock API to query analyses and implement optimizations.

4.1 Control-Flow Specification

The first component that is required when using Flock for a language implementation is the control-flow graph definition. This serves as the basis for writing analyses in FlowSpec, since the data-flow rules refer to sorts and constructors defined in this definition. The CFG definition in FlowSpec consists of several sections. The full CFG definition can be found in Appx. A.

4.1.1 Signatures

The first section, shown in Fig. 4.1, contains signatures of the subject language, consisting of the sorts and constructors. These sorts and constructors mirror the syntax definitions for Tiger in SDF3, since the terms produced by parsing with SDF3 must match the terms expected by Flock. The signatures defined in this section can be used in the next section to define the CFG semantics.

4.1.2 Control-Flow Rules

The second section consists of the control-flow rules that dictate how the AST should be turned into a CFG. Fig. 4.2 contains some example rules from the Tiger language. The example contains the three variants of control-flow rules: a root rule, a node rule, and regular rules. The semantics of these rules are explained in Ch. 2.

4.2 Data-Flow Specification

Following our CFG definition we can now implement the necessary analyses for our optimizations. An analysis consists of one or more property definitions, the property rules that specify the transfer functions of the analysis, and optional type and function definitions. We show three different types of analyses: liveness, value, and array length analysis with different types of flow functions and abstract domains.

```

module sorts
signature
  sorts
    Id = string
    Var
    Exp
    // ...

  constructors
    VarDec : Id * Type * Exp -> Dec

    Eq : Exp * Exp -> Exp
    Times : Exp * Exp -> Exp
    If : Exp * Exp * Exp -> Exp

    Occ : Id -> Occ
    // ...

```

Figure 4.1: Declaration of AST sorts and constructors in FlowSpec

```

control-flow rules
  root ProcDec(n, args, body) = start
                                -> body -> end
  ProcDec(_, _, _) = entry -> exit

  VarDec(n, t, e) = entry -> e
                    -> this -> exit
  Eq(lhs, rhs) = entry -> lhs -> rhs
                 -> this -> exit
  Times(lhs, rhs) = entry -> lhs -> rhs
                    -> this -> exit
  If(c, t, e) = entry -> c -> t -> exit,
               c -> e -> exit

  node Var(_)
  //...

```

Figure 4.2: Declaration of CFG rules in FlowSpec

Property Definition

The property is the lattice type computed for each program node. A property declaration consists of a name and a lattice type. The lattice type can be a builtin lattice type such as a `may/must` set or map type, or a user-defined lattice type.

Property Rules

The property rules, also called *transfer functions*, define how the property is computed. These rules use the signatures defined in the `signature` section to pattern match on AST terms. For each AST term, the first rule with the first pattern to match is executed.

Type Definitions

A dataflow analysis can also include definitions for types as shown in Fig. 4.6. There are two kinds of types: datatypes and lattice types. These type definitions allow a language developer to specify the abstract domain and lattice operations for their analyses.

```

properties
  live: MaySet[string]

```

Figure 4.3: Property definition for liveness analysis for Tiger in FlowSpec

```

property rules
  live(_.end) = {}
  live(VarDec(n, _, _) -> next) =
    {m | m <- live(next), m != n}
  live(Var(n) -> next) =
    {n} \/ live(next)
  // ...
  live(_ -> next) = live(next)

```

Figure 4.4: Dataflow rules for liveness analysis for Tiger in FlowSpec

4.2.1 Liveness

Fig. 4.3 and Fig. 4.4 show the property definitions and rules for liveness analysis. Liveness analysis computes the set of live variables at every program point. A variable is live at a

program point if its value at that time may be used later. Inversely, a dead variable holds a value that is never read. The definition of liveness does not include custom type definitions since the builtin types of `MaySet` and `string` suffice.

The property rules specify that a `VarDec` removes the assigned variable from the set of live variables, while a `Ref` adds to it. We omit some property rules for simplicity, such as the rules for loop bindings. The full specification can be found in Appendix B.

properties

```
values: SimpleMap[string, Value]
```

property rules

```
values(_.end) = {}
```

```
values(prev -> VarDec(n, _, Int(i)))
  = { k |-> v | (k |-> v)
      <- values(prev), k != n }
  \/ {n |-> Const(i)}
```

```
values(prev -> VarDec(n, _, _))
  = { k |-> v | (k |-> v)
      <- values(prev), k != n }
  \/ {n |-> Top()}
```

```
// ...
```

Figure 4.5: Property definition and rules for value analysis for Tiger in FlowSpec

types

```
ConstProp =
```

```
| Top()
```

```
| Const(int)
```

```
| Bottom()
```

lattices

```
Value where
```

```
type = ConstProp
```

```
bottom = Bottom()
```

```
top = Top()
```

```
lub(l, r) = match (l, r) with
```

```
| (Top(), _) => Top()
```

```
| (_, Top()) => Top()
```

```
| (Const(i), Const(j)) =>
```

```
  if i == j then (Const(i)) else (Top())
```

```
| (_, Bottom()) => l
```

```
| (Bottom(), _) => r
```

Figure 4.6: Type definitions for value analysis for Tiger in FlowSpec

4.2.2 Value

Value analysis approximates the value corresponding to each variable at each program point. The abstract domain of our value analysis consists of a top value, bottom value, and an integer constant. This data type is defined in the `types` section in Fig. 4.5, and is called `ConstProp`. It is then used in the `Value` lattice definition, which defines the necessary lattice operations on this abstract domain.

The property rules in Fig. 4.6 specify that a variable declaration with a constant right-hand side is stored in the lattice with a `Const` value, whereas any other declaration is stored as `Top`, indicating that the analysis cannot derive useful information about the value of the variable. The full definition can be found in Appendix B.

4.2.3 Array Length

To provide diagnostics such as bounds-checking as editor service, we implement a simple array length analysis as shown in Fig. 4.7. The property rules for this analysis are similar to value analysis, but instead of recording values stored in variables we record the lengths of array values. Again we omit some of the rules which can be found in Appendix B. We show how this analysis can be used for diagnostics in Section 4.3.3.

4.3 Queries, Optimizations, and Diagnostics

In this section we will show how the analysis definitions of the previous section can be used to perform analysis queries for optimizations and diagnostics.

```

properties
  lengths: SimpleMap[string, Value]

property rules
  lengths(_.end) = SimpleMap[string, Value].bottom

  lengths(prev -> VarDec(n, _, Array(_, Int(i), _)))
    = { k |-> v | (k |-> v) <- lengths(prev), k != n } \\/ {n |-> Const(i)}

  lengths(prev -> VarDec(n, _, _))
    = { k |-> v | (k |-> v) <- lengths(prev), k != n } \\/ {n |-> Top()}

  // ...

```

Figure 4.7: Property definition and rules for array length analysis for Tiger in Flowspec

```

external flock-get-live()
external flock-live-contains(|key)

external get-values()
external get-values(|key)

external get-lengths()
external get-lengths(|key)

```

Figure 4.8: Generated Stratego API for liveness, value, and array length analysis

4.3.1 Query API

After compiling the analyses and incorporating the built output in our project, we can query the analyses using the generated Stratego-Flock API shown in Fig. 4.8 and use the results for optimization.

```

pipeline = flock-initialize; flock-fixpoint(pass|3)

pass = onctd(propagate-constant); onctd(remove-dead-vardec)

propagate-constant: lv@LValue(v@Var(n)) -> <flock-replace-node(|lv)> Int(c)
  where
    n' := <strip-annos> n
    ; Const(c) := <flock-get-values(|n')> v

remove-dead-vardec: a@VarDec(n, _, _) -> <flock-replace-node(|a)> Hole()
  where
    <not(flock-is-live(|n))> a

```

Figure 4.9: Optimization pipeline for Tiger in Stratego

4.3.2 Optimization Pipeline

Shown in Fig. 4.9 is a high-level example of an optimization pipeline for Tiger. Given a Tiger program p , we initialize the runtime, and proceed to execute two optimization passes that first propagates constant values and then removes dead variables. We execute these passes three times or less if we reach a fixpoint.

Dead variables are removed by replacing them with `Hole` terms as we cannot efficiently remove elements of a list of terms due to a limitation in the Flock API. These `Hole` terms can be efficiently removed in a later stage.

```
editor-analyze: (ast, path, project-path) -> (ast', [], warn, [])
where
  ast' := <flock-initialize> ast
; warn := <collect(out-of-bounds); map(make-message(|"Out of bounds"))> ast'

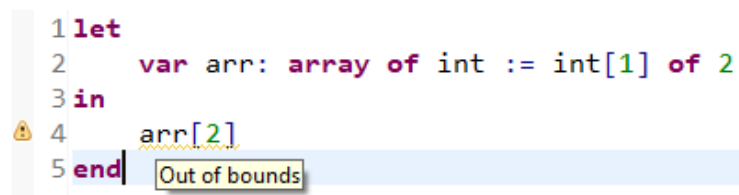
out-of-bounds: s@Subscript(LValue(v@Var(n)), Int(i)) -> s
where
  n' := <strip-annos> n
; x := <flock-get-lengths(|n')> v
; Const(l) := x
; i' := <string-to-int> i
; <string-to-int; int-leq(|i')> l

make-message(|m): subscript -> (subscript, m)
```

Figure 4.10: Diagnostics that report out-of-bounds array accesses

4.3.3 Editor Diagnostics

We can also use Flock to provide editor diagnostics to a language user. Fig. 4.10 shows a Stratego implementation of out-of-bounds diagnostics using the array length analysis. The implementation defines the `editor-analyze` strategy, which is a hook for adding diagnostics to Spoofox. This strategy is invoked once when a file is loaded. The strategy initializes the runtime and then finds all array accesses that are known to be out-of-bounds by comparing the index to the analysed array length. An example of a program with an out-of-bounds access and the resulting warning can be seen in Fig. 4.11. This program defines an array with a single element but attempts to access the third element.



```
1 let
2   var arr: array of int := int[1] of 2
3 in
4   arr[2]
5 end
```

Figure 4.11: Example of out-of-bounds diagnostics reported by array length analysis

```
editor-hover: (v@LValue(_), _, _, _, _) -> val
where val := <get-analysis-value + !Top()> v

get-analysis-value: lv@LValue(v@Var(n)) -> <flock-get-values(|n')> v
where
  n' := <strip-annos> n
```

Figure 4.12: Implementation of hover tooltips to show value analysis results

Finally, we use Flock to provide value information on hover. This means a user can hover over a variable and inspect the dataflow information of that variable. Fig. 4.12 shows the implementation of a simple value analysis result on hover. Similar to `editor-analyze`, `editor-hover` is a hook to provide Spoofox with hover information. This strategy is given the

AST node being hovered over, invokes the value analysis, and returns the result. Spoofox/Eclipse then renders the result in the editor. The results can be seen in Fig. 4.13

```
1 let
2   var x: int := 3
3 in
4   x|
5 end Const("3")
```

Figure 4.13: Example of on-demand dataflow analysis in an editor

Chapter 5

Solution Implementation

In this chapter we will discuss the implementation of the Flock compiler, the Flock Stratego API, and the Flock runtime. For each of these we will provide an overview of the components, and important datastructures and algorithms we used.

5.1 Flock Compiler

The Flock compiler has two compilation pipelines. The first processes the control-flow rules in a FlowSpec declaration into a Java class that turns Tiger programs into the runtime's control flow graph structure, called the *CFG Builder*. The second processes the dataflow rules in a FlowSpec declaration into the Java classes that perform the specified analysis.

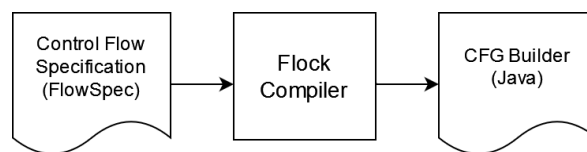


Figure 5.1: Output file of the Flock compiler when compiling a FlowSpec Control Flow specification

As shown in Fig. 5.1, the compiler can compile a FlowSpec file containing control-flow rules into a CFG builder class. This class transforms a program into its CFG for use in dataflow analysis. This is a separate compilation pipeline since the user only needs to specify these control-flow rule and generate the CFG builder once, regardless of the number of analyses.

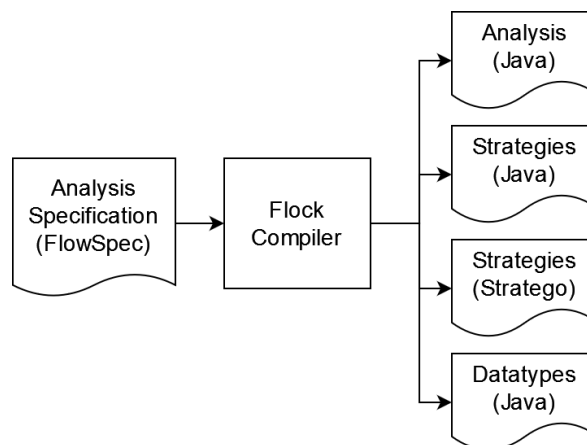


Figure 5.2: Output files of the Flock compiler when compiling a FlowSpec analysis

Shown in Fig. 5.2 are the various artefacts generated by the compiler for an analysis written in FlowSpec. The *analysis* file contains definitions of transfer functions, pattern matching logic to assign transfer functions to CFG nodes, lattice datatypes, and user functions. The *datatypes* file contains other datatypes. The *strategies* files contain the implementation of the strategies exposed to the user to query the analysis results.

5.1.1 Compilation Pipeline

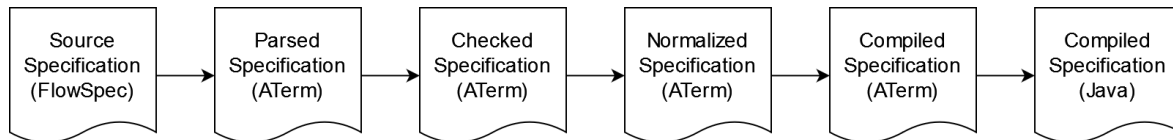


Figure 5.3: Pipeline stages of the Flock compiler

The process of compilation from FlowSpec declaration to Java classes consists of several phases, shown in Fig. 5.3.

The compilation process starts in the parsing stage. This phase is implemented in the *SDF3* metalanguage, which takes a grammar specification and gives us a parser. This parser outputs the FlowSpec declaration in ATerm format, which we can process using *Stratego* in the following phases.

After parsing the program is typechecked using a specification written in *Statix*. This phase ensures that our program is well-typed, and moreover exposes type information to further phases in the compilation pipeline.

The next phase, called normalization, performs many small transformations that simplify the code generation phase. First of all, we rename variables to ensure uniqueness compared to autogenerated names. Then we normalize various expressions, including hoisting complex expressions into let bindings. Finally we embed type information explicitly into the AST, so that the compiler does not need to interface with *Statix* during the code generation phase. This type information exists in the AST in the form of explicit casts in favour of implicit conversions, typed lattice operations, typed let bindings, etc.

The final phase is code generation. This phase is the only phase that differs between the various output files. The most complex implementation performs the code generation for the analysis file. It turns the type-annotated FlowSpec code into their corresponding Java components. The final step of this phase is pretty-printing the generated code, which is in ATerm format, to a Java file.

5.1.2 Runtime Integration

After compilation, the generated files are integrated into the runtime. Fig. 5.4 shows an overview of the dependencies between the optimization pipeline, generated components, and standard components. The user can invoke either generated strategies that represent analysis queries, or the user can invoke standard strategies to communicate program changes. All of these strategies interface with the runtime. The runtime in turn depends on the generated analyses or CFG builder classes.

5.2 Flock Stratego API

The Flock Stratego API is the set of Stratego strategies (analogous to functions in other languages) that are available to the user. They are the interface between the optimization pipeline written in Stratego and the runtime written in Java. The API consists of a set of common strategies, and strategies generated from the analyses.

- `flock-initialize()`: Initializes the Flock runtime, clearing any previous instance data.
- `flock-remove-node()`: Removes the given node from the Flock program model.

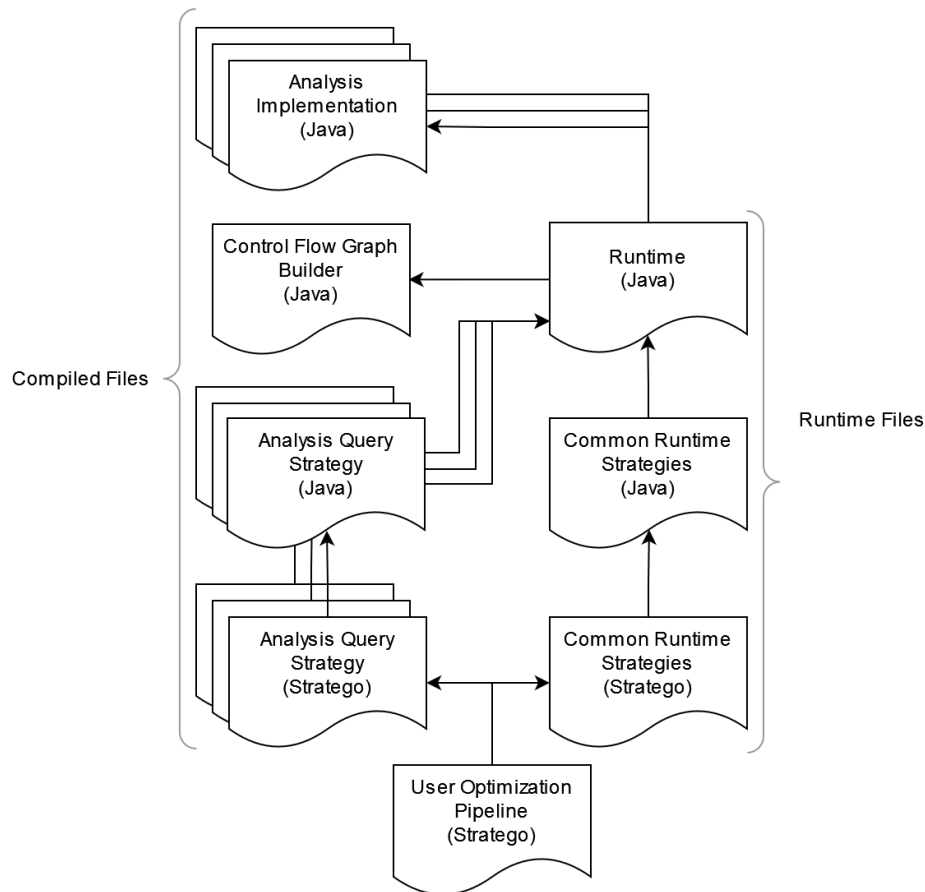


Figure 5.4: High-level overview of the runtime component structure

- `flock-remove-all(filter|)`: Removes every node within the given list for which `filter` succeeds.
- `flock-replace-node(|old-node)`: Replaces the `old-node` with the given node in the Flock program model.
- `flock-make-id(|)`: Adds the required `FlockNodeId` to each (sub)term in the given term.
- `flock-fixpoint(strategy|bound)`: Runs `strategy` on the given term to a fixpoint, bounded to `bound` iterations.
- `flock-debug-graph(|property)`: Prints a `.dot` representation of the Flock program model, with the `property` values for each node added. This allows for easy visualization using Graphviz (Ellson et al. 2001).

The strategies generated for an analysis depend on the type of the property it computes. The following strategies are available for an analysis with a property named S of type Set:

- `flock-get-S(|)`: Returns the computed set at the location of the given node.
- `flock-S-contains(|name)`: Succeeds if the name string is present in the computed set at the location of the given node. This strategy avoids synthesizing a Stratego list.

The following strategies are available for an analysis with property named M of type Map:

- `flock-get-M(|)`: Returns a list of (name -> value) pairs that correspond to the computed map at the location of the given node.

- `flock-get-M(|key)`: Returns the value corresponding to the given `key` at the location of the given node. This strategy avoids synthesizing a Stratego list.

Other types of properties, those that do not produce a Set or Map result, compile only to a generic *flock-get-property*.

A minimal working optimization pipeline using Flock will need to invoke `flock-initialize` and `flock-make-id` once, followed by invocations of the analysis query strategies coupled with program edit strategies such as `flock-replace-node`. The `flock-fixpoint` strategy serves as a convenience strategy to repeatedly apply a set of optimizations that may unlock new optimizations within that set. It also allows the user to bound the number of iterations.

5.3 Flock Runtime

The runtime provides the user with on-demand incremental analysis results. In this section we will discuss the implementation of the runtime, including the incrementalization of the analyses. We start by describing at a high-level the steps performed by the runtime after 1) a transformation is performed and 2) an analysis is queried. We will then describe these steps in more detail, and discuss the algorithms and datastructures involved. Finally we will discuss the class structure of generated analyses and the role of the runtime in providing functionality to these analyses.

5.3.1 High-Level Overview

In the simplest terms, the runtime is able to incrementally run an analysis when the optimizer queries it, and it can incrementally update its datastructures when a program transformation is performed. Furthermore, the runtime contains an initialization phase during which its internal datastructures are initialized. An overview of each of these steps are shown in Figures 5.5, 5.6, and 5.7.

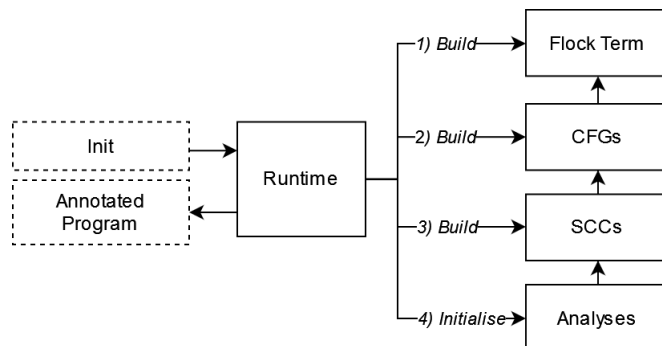


Figure 5.5: Steps performed during initialization

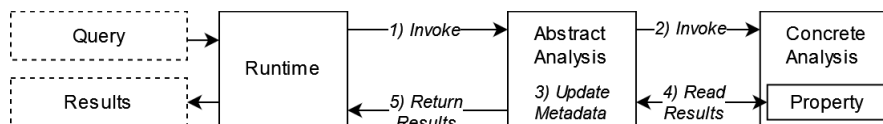


Figure 5.6: Steps performed after an analysis is queried

Flock IDs

Due to the immutability of term in Stratego, terms of a program are often copied, meaning we cannot use referential equality to determine to which part of the program a given term refers. For instance, a term such as `Add(Int("1"), Int("1"))` may appear twice in a program. When the optimizer invokes the `flock-replace-node` strategy, replacing this term with `Add("2")`, it is not clear which of the two occurrences is being replaced. For this reason we annotate each

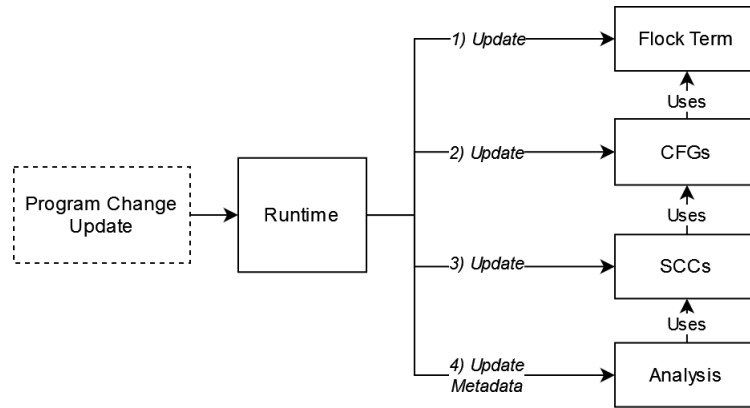


Figure 5.7: Steps performed after a transformation is performed

term in the program with a unique identifier, the Flock ID. Additionally, whenever a term is replaced in the tree we annotate each of its (sub)terms with new identifiers. This gives us a way of uniquely identifying any term in the program under analysis. These identifiers are used wherever we need to address a term uniquely, such as the mapping between CFG nodes and terms of our program.

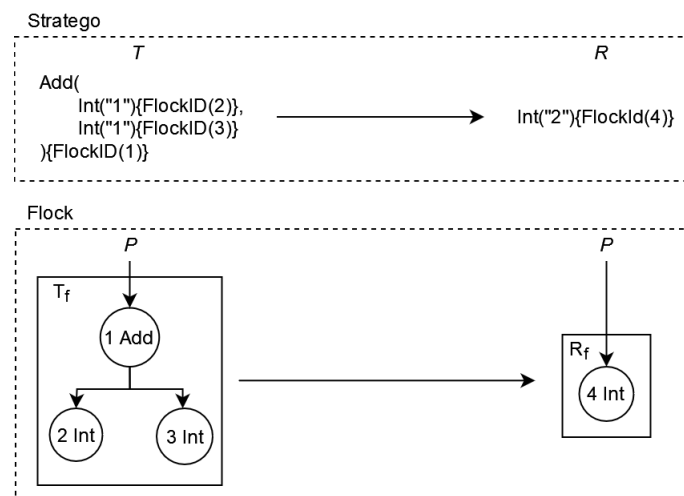
5.3.2 Transformations

When a transformation is performed the runtime must incrementally update its internal datastructures as shown in Fig. 5.7, so that an analysis can be performed if it is queried. These datastructures depend on each other in a linear order, from Flock term to CFG, to the SCC partitioning of the CFG and their stratification. We will describe what these datastructures are, why they are necessary, and how they are updated.

Flock term

The Flock term is a mutable mirror of the program under analysis. The runtime uses this Term Tree during CFG construction and analysis instead of a Stratego term representing the program. The need for the Term Tree is explained in more detail in Section 5.4.

Initially, the Flock term is constructed from the Stratego term passed to the `flock-initialize` strategy. The Flock term class hierarchy and structure mirrors that of the Stratego term, so mirroring changes to the Stratego term is straightforward.

Figure 5.8: Replacement of term in Stratego term and Flock term. Note that P is unchanged in the Flock runtime, so that references into P remain valid.

Replacement Replacement of a Stratego term translates to a replacement of a subtree of in the Flock term. Given a Stratego term T and its replacement Stratego term R in a program P , we find the Flock term T_f that has the same Flock ID as T and remove it from the Flock term. We create a new Flock term R_f from R , and add it to P . Finally we set the parents of R_f to be the previous parents of T_f , if they exist.

Deletion Deletion of a Stratego term is a simple deletion of a subtree in the Flock term. Similar to the case of replacement we use the Flock ID of T to find its corresponding subtree and remove it.

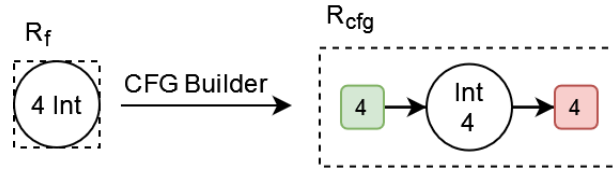


Figure 5.9: Creation of CFG from a Flock term. Green nodes are entry nodes, red nodes are exit nodes.

Control-Flow Graph

The Control-Flow Graph (CFG) is a graph datastructure that explicitly encodes the control-flow of the program in its edges and nodes. In this section we will discuss the design of the CFG datastructure and the incremental updates that we can perform on it.

The design of the CFG in Flock is a slightly more complex variation on the CFG discussed in Ch. 2.2. The difference is that CFGs in Flock have explicit entry, exit, start, and end nodes, mirroring their use in FlowSpec.

A start node functions solely as a root node, and similarly an end node functions solely as a leaf node. Other nodes in the graph can not be a root or leaf node.

Entry and exit nodes are explicit in our CFG because they greatly simplify incremental updates to the graph which we describe in the following sections. Together with the CFG we maintain a mapping from Flock terms to each of these special nodes.

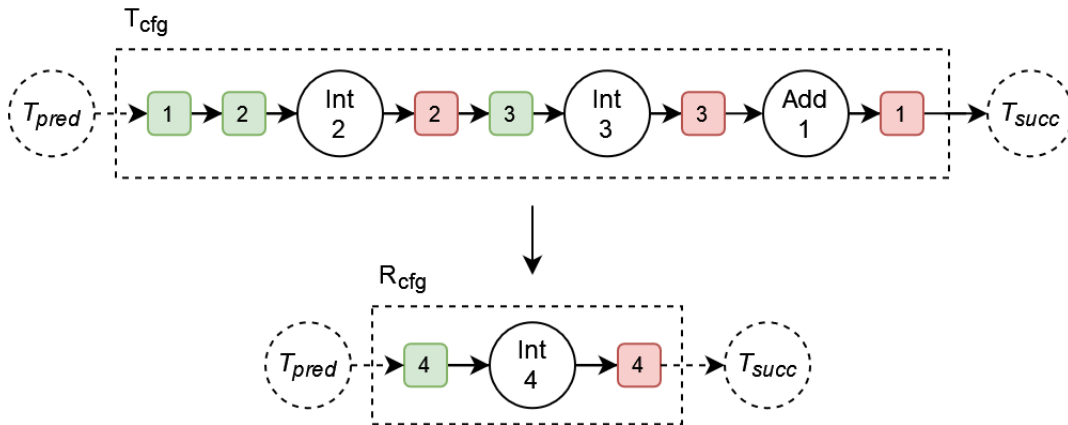


Figure 5.10: Replacement of a (sub-)CFG

Replacement The CFG builder transforms a Flock term node into a CFG. Given the same Stratego terms T , R , and P as before, we use the builder to create R_{cfg} , the CFG of R as shown in Fig. 5.9. We then replace the existing T_{cfg} with R_{cfg} in the program CFG as follows:

Let T_{entry}/T_{exit} be the entry/exit node corresponding to the root term of T . We call the nodes in P_{cfg} with edges to T_{entry} the predecessors of T_{cfg} , or T_{pred} . Similarly we call the nodes in P_{cfg} with edges from T_{exit} the successors of T_{cfg} , or T_{succ} .

We remove the edges between T_{pred} and T_{entry} and create edges between T_{pred} and R_{entry} . Similarly we replace the edges between T_{exit} and T_{succ} with edges between R_{exit} and T_{succ} . An example of this replacement can be seen in Fig. 5.10.

As part of the removal of T_{cfg} we also remove each associated special node from P_{cfg} .

Deletion In the case of deletion R_{cfg} will be empty. We create edges from each node in T_{pred} to each node in T_{succ} to bridge this hole in the CFG.

Partitioning CFG into Strongly Connected Components

We initially compute the strongly connected components (SCCs) of the CFG in topological order with Tarjans algorithm (Tarjan 1972). The SCCs partitioning of the CFG gives the order in which nodes must be processed during analysis. It also gives us a way to categorize which nodes are up-to-date and which must still be analyzed, which we will describe in the next section.

We incrementally update the SCCs at the same time as the CFG. We will describe the procedure for doing so, assuming the same T , R , and P as before. Our implementation maintains a set of components, a bidirectional mapping between components, and a mapping from nodes to components. Given this last mapping from a node to its component, we find the set of components C_{sT} that contain at least one of the nodes in T . Similarly, we compute the sets of components $C_{s_{pred}}$ and $C_{s_{succ}}$ for the set of predecessors and successors of T , computed earlier.

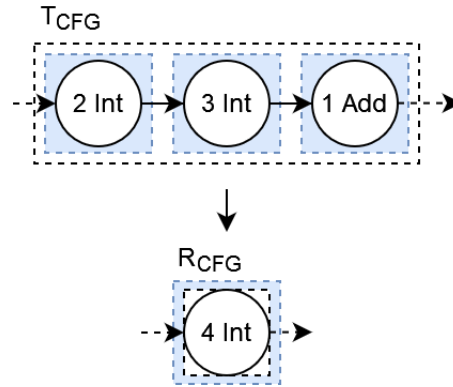


Figure 5.11: Resulting SCCs after a replacement with no cycles present.

We distinguish two cases based on the structure of the CFG. The first case, as shown in Fig. 5.12, is when there is a common component C in $C_{s_{pred}}$ and $C_{s_{succ}}$. This implies that there is a path from the leafs of T_{cfg} back to its roots, outside of the T_{cfg} itself. When we replace T_{cfg} with R_{cfg} , this path will be unchanged (an assumption discussed in the *Limitations*), and thus R_{cfg} will be part of C . No components are created or destroyed.

The second case is when there is no common component between $C_{s_{pred}}$ and $C_{s_{succ}}$ as shown in Fig. 5.11. This implies that the components C_{sT} only contain the nodes in T (an assumption similarly discussed in the *Limitations*). Therefore we compute the SCCs formed by R_{cfg} in isolation and can assume the components are also SCCs in P .

Limitations

The procedure described above is an efficient way to incrementally maintain the SCCs of a program, but it is also limited in the types of control flow constructs it can support due to several assumptions made about the structure of the CFGs. This procedure is only correct if the control-flow rules in the FlowSpec specification a) only refer to surrounding CFG nodes through the *entry* and *exit* constructs, and b) there always exists a path from the *entry* to the *exit* node. In other words, there is no limitation on how the CFG nodes are constructed and connected locally (within the scope of a single term), but they cannot be connected to surrounding CFG nodes arbitrarily. In practical terms this means that non-local control-flow

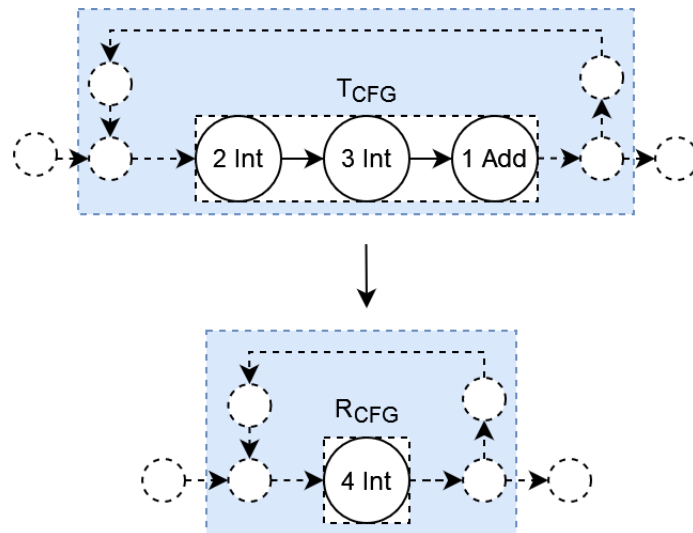


Figure 5.12: Update to SCCs after a replacement within a cycle.

constructs such as *goto*, *exceptions*, etc. cannot be supported, as they may affect the SCCs of the CFG in more complex manners.

As a fallback, Flock will mark terms that do not meet the constraints as *irregular*. When an irregular node is created or replaced, Flock will recompute the SCCs of the entire program. This of course incurs a performance penalty that we do not measure as Tiger does not support such constructs. Many other languages make do support these more complex control-flow constructs, so this presents a threat to validity of our experiments which we discuss in more detail in Ch. 6.4.3.

Stratification of SCCs

Our last step is the division of the SCCs into two groups: clean and dirty. A clean SCC only contains nodes with fully up-to-date analysis results. Analysis results in dirty SCCs are considered outdated. By the definition of SCCs and dataflow, we know that a result computed at a node n in SCC C can propagate to any other node in C , and any other SCC reachable from C . The same holds for the invalidation of analysis results, since if the results at a node n may propagate to a node m , a change in n implies a possible change in results at m . Given a change to an SCC C , we mark C and all other SCCs reachable from C as dirty with a simple depth-first traversal. This means they must be reanalyzed before their results can be used.

Furthermore, we maintain a set of *new* nodes. We initialize our worklist only with these new nodes, since any changes to the analysis results must come from them.

5.3.3 Queries

Queries are incrementalized by using the SCCs as explained in Section 5.3.2. Given a query at node n in SCC C , we must initialize our worklist with every *new* node that may reach n . Every node in a dirty SCC that can reach C (including C itself) *may* be added to the worklist during iteration. All other nodes in the CFG are skipped. After analysis is complete we mark each of these SCCs as clean, since we know they have been fully analyzed.

Worklist Algorithm

The worklist algorithm performs the actual analysis. It is based on the regular worklist algorithm from Chapter 2, Section 2.2. The algorithm is shown in pseudocode in Fig. 5.13. Given a query at node n that is part of component C , the worklist algorithm is invoked for each component for which C is reachable, in topological order.

This worklist algorithm differs from the standard algorithm in a few ways. First are the loops before our main loop. The first loop remains unchanged. The second loop computes an

initial dataflow solution at the `start` nodes to bootstrap the analysis. The third loop pulls in dataflow information from predecessor components, as we do not propagate this information across component boundaries during the main loop.

The main loop is of the worklist algorithm pops a node `v` off the queue and propagates dataflow information to its successors. This makes it easy to ignore predecessor nodes outside of the current component in our main loop.

Finally we need to update the status of our component for future queries by marking them as clean.

```

worklist = {}
for v in C:
    initialize v
    add v to worklist

for v in C:
    if v is Start node:
        compute initial solution at v

for v in C:
    compute solution at v from predecessors

while len(worklist) > 0:
    v = worklist.pop

    for s in successors of v:
        if s not in C:
            continue

        recompute solution at s
        if the solution changed:
            add successors to worklist

mark C clean
mark nodes in C as clean

```

Figure 5.13: Adapted worklist algorithm used in the Flock runtime

5.3.4 Analysis Class Structure

The runtime provides a set of common classes to reduce the complexity of the generated analyses and the compiler. Fig. 5.15 shows these classes in relation to an analysis implementation. The `Analysis`, `Lattice`, `Value`, `TransferFunction`, and `Property` classes are abstract base classes that provide functionality to the analysis, and keep the runtime generic with regards to analysis implementations. The `Helpers` and `Utils` classes provide common functionality for dealing with Stratego terms, `Set/Map` types, etc. in a lattice-aware manner.

5.3.5 Avoiding Copies with the Capsule Library

A common pattern in dataflow analyses written in FlowSpec is to have a rule that adds or removes a single entry from a map or set. Consider for example the rule shown in Fig. 5.14. This rule applies to a `VarDec` node with a constant RHS. It specifies that the resulting lattice is a copy of its input with a single entry replaced.

A naive implementation of this dataflow rule may create a distinct map for each control-flow graph node, even though all but one entry is unchanged. A more efficient implementa-

```

values(prev -> VarDec(n, _, Int(i)))
  = { k |-> v | (k |-> v) <- values(prev), k != n } \ / {n |-> Const(i)}

```

Figure 5.14: Example of a dataflow rule that copies a map and changes a single entry

tion can share the entries of the map that are unchanged, while separately recording unique entries. This decouples the cost of executing this dataflow rule from the size of the map. We implement this optimization by using efficient immutable collections (Steindorfer 2017).

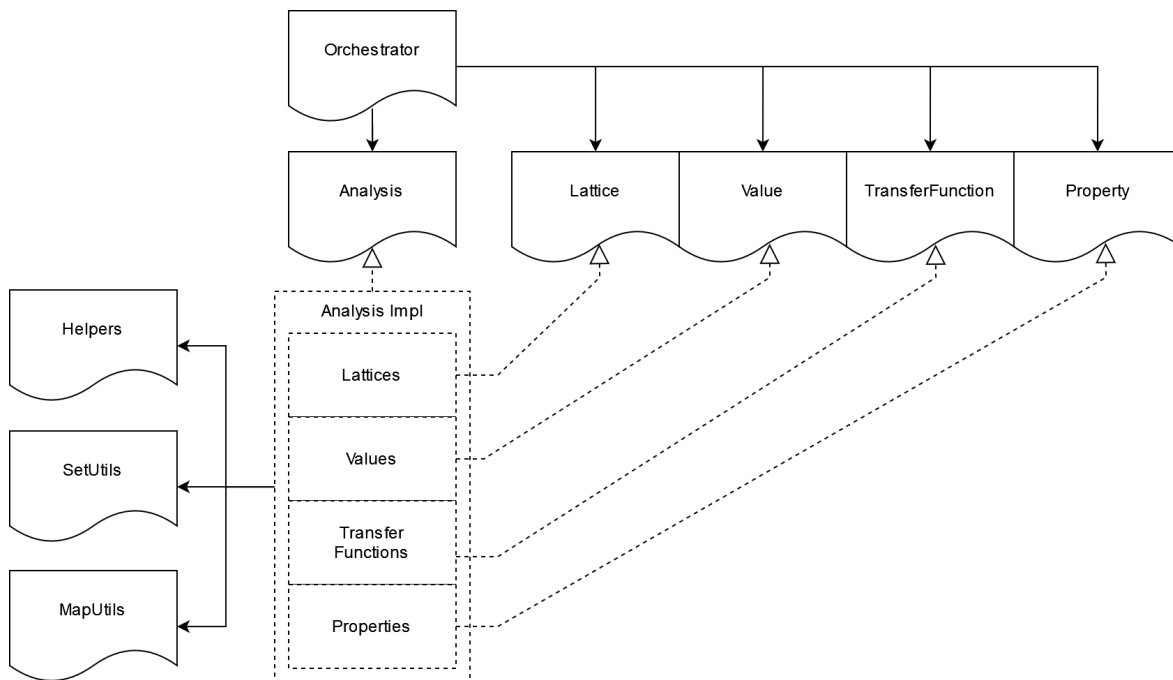


Figure 5.15: Class diagram of a compiled analysis.

5.4 Incremental Analysis in an Immutable Language

One of the key challenges we encountered while implementing Flock was due to the immutability of Stratego terms.

The program under analysis lives in the user environment, since we do not want to restrict what the user can do in the optimization pipeline. The runtime must also have access to the program, as analyses use terms (such as numbers, string, or expressions) of the program in the flow functions and during pattern matching. The runtime must maintain a view of the program that it can use during analysis, because the changes communicated to the runtime are separate from the changes being applied to the actual program term. This is because several optimizations may be chained together before the optimized program is actually synthesized.

The challenge lies in providing the analyses with an up-to-date view of the program, while the input to the runtime are diffs. Since the analyses are not designed to operate on a sequence of diffs, we considered two options. Either we redesign parts of the system to implement analyses on sequences of diffs directly, or we avoid this complexity and translate diffs into mutations of an internal representation of the program, so that the analyses can remain mostly the same. We chose the latter option since we do not know the impact on complexity and performance of the first option.

The implemented solution translates diffs into mutations on a separate representation of the program. This internal datastructure is called the *TermTree* in e.g. Fig. 5.7. It is homomorphic to the Stratego representation of a program, but allows mutation of subterms. This means references pointing inside the tree (used for e.g. pattern matching) do not need to be updated when a subterm of the pointee is changed. Each CFG node contains a reference to

the term from which it was created. Importantly, this implies there are no invalid references left after a program change as the CFG is updated together with the TermTree, and no other references pointing inside the tree exist.

The analysis results live within the environment of the runtime, so can be mutated directly by the analysis classes. The API was not designed to give the user diffs of the analysis results, so this did not introduce any problems. When the optimizer queries an analysis result, we build a Stratego term representing the result. Due to the performance penalty and inconvenience of producing large maps or sets we generate additional strategies described in section 5.2 that can be used to avoid this penalty in many cases.

Chapter 6

Evaluation

In this chapter we discuss our evaluation of the Flock project. The evaluation of the project consists of several parts. First, we implement and run an optimization pipeline on several families of programs and measure the run times. Second, we measure the memory usage of these same optimization pipelines. Finally, we measure the latency of random analysis queries without program edits similar to how a user may interact with the editor.

6.1 Run Time Analysis of an Optimization Pipeline

We measure the run time of three different optimizers: Flock, Dynamic Rules, and LLVM. The Flock-based optimizer is our own implementation. The Dynamic Rules optimizer is also implemented with Spoofox and is adapted from Bravenboer et al. (Bravenboer et al. 2006). The LLVM optimizer is a well-known and widely used optimizer (C. A. Lattner 2002).

6.1.1 Benchmarking Methodology

We benchmark the run time of the three optimizers by synthesizing various programs that can be optimized to a single constant value. We chose five different types of programs that differ across the following characteristics: size and shape of the generated control-flow graph, types of transformations required to optimize, and the number of (nested) scopes introduced. We chose these characteristics as we believed these would each provide valuable insight into the performance of Flock. The number of nested scopes is a characteristic that is known to impact the performance of the Dynamic Rules optimizer (Bravenboer et al. 2006), and is thus an interesting characteristic to measure against the Flock optimizer.

For each type of program we generate instances ranging from small sizes up to $n = 5000$, to require a long enough run time for significant results.

We test the Flock and Dynamic Rules optimizers using JMH (*JMH: Java Microbenchmark Harness* 2022) to achieve repeatable results. Both optimizers expose a Stratego strategy that takes a program term and returns its optimized equivalent. For each program, we measure the time taken for these strategies to execute.

We test the LLVM optimizer by generating equivalent c-lang versions of our test programs. We compile these to LLVM bitcode using Clang, with optimizations disabled. We then run and measure the time taken by LLVM to optimize these bitcode files to ensure we do not include the translation from c to LLVM bitcode in our measurements.

6.1.2 Benchmark: Vars

The first benchmark is a simple chain of variable definitions and variable uses. An example of a generated program is shown in Fig. 6.2. This program is generated for various instances of n , up to $n = 5000$. The run time of the optimizers for each n is shown in Fig. 6.1. We can see that Flock is significantly slower than the other implementations. The performance of LLVM may be explained by their use of single-static assignment form, which makes it possible to optimize this type of program without dataflow analysis as the entire program fits within a single basic-block. The performance difference with DR may be caused by the

extra cost incurred by building and updating the control-flow graph. This means DR gains performance at the cost of flexibility, as the order of strategy applications must match the control-flow of the program.

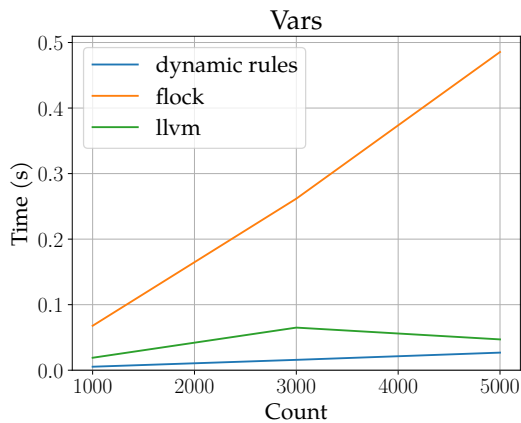


Figure 6.1: Run time of optimization pipelines on various sizes of the program shown in Fig. 6.2

```

let var a0: int := 1
    var a1: int := a0 + 1
    var a2: int := a1 + 1
in
    a2
end

```

Figure 6.2: Example program for $n=3$

6.1.3 Benchmark: Branches

The second benchmark is a variation of the previous by adding branching logic to each variable declaration, as shown in Fig. 6.2. The run time of the optimization pipeline for each n is shown in Fig. 6.3. We see that Flock takes roughly twice the time compared to the previous benchmark, matching the increase in size of the CFG. LLVM shows a non-linear run time, and is quickly outperformed by Flock. This matches our expectations as this program contains many basic blocks in its LLVM bitcode representation, requiring the more expensive dataflow analysis to optimize.

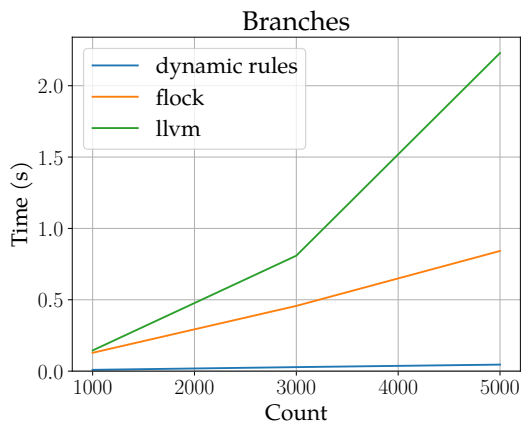


Figure 6.3: Run time of optimization pipelines on various sizes of the program shown in Fig. 6.4

```

let var a0: int := 1
    var a1: int := if a0 > 0 then a0 else 0
    var a2: int := if a1 > 0 then a1 else 0
in
    a2
end

```

Figure 6.4: Example program for $n=3$

6.1.4 Benchmark: Nested Scopes

The third benchmark introduces nesting of scopes with variable accesses to the outermost scope, as shown in Fig. 6.6. The run time of the optimization pipeline for each n is shown in Fig. 6.3. We see that the Dynamic Rules implementation slows down significantly due

to the distance between scopes of variable accesses and their declarations. This is a known characteristic of this implementation (Bravenboer et al. 2006).

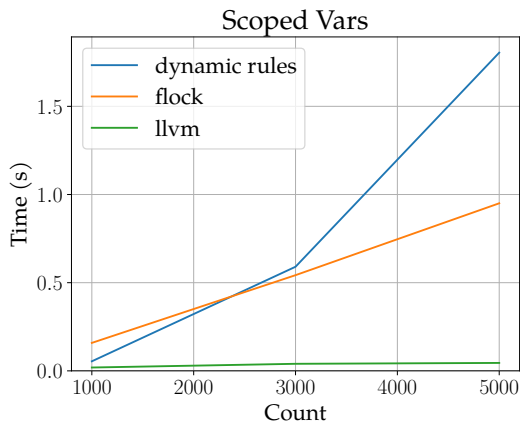


Figure 6.5: Run time of optimization pipelines on various sizes of the program shown in Fig. 6.6

```

let var a0: int := 1 in
  let var a1: int := a0 + a0 + 1 in
    let var a2: int := a1 + a0 + 1 in
      a2
    end
  end
end

```

Figure 6.6: Example program for $n=3$

6.1.5 Benchmark: Inlining (Linear)

The fourth benchmark requires function inlining for optimization, shown in Fig. 6.8. We see that Flock is slowest, similar to the first benchmark, but with similar linear growth as Dynamic Rules and LLVM.

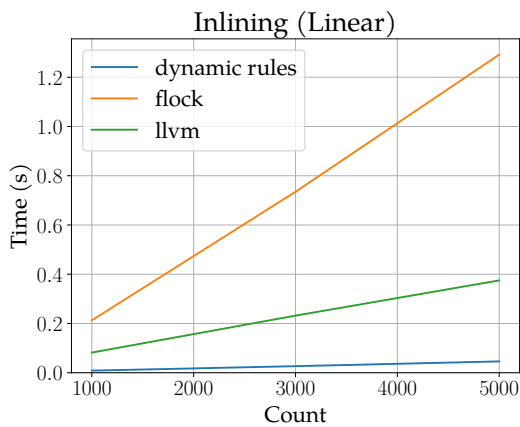


Figure 6.7: Run time of optimization pipelines on various sizes of the program shown in Fig. 6.8

```

let function a0(a: int): int = a + 1
  function a1(a: int): int = a + 1
  function a2(a: int): int = a + 2
in
  a0(1) +
  a1(1) +
  a2(1)
end

```

Figure 6.8: Example program for $n=3$

6.1.6 Benchmark: Inlining (Recursive)

The fifth benchmark similarly requires function inlining for optimization, but in a recursive manner such that each inlining transformation leaves a new function call to inline (except the leaf function). We see that Flock cannot efficiently optimize (large) programs with this structure.

This performance issue is the result of a lack of flexibility offered by the Flock API. During the optimization of the program we perform three types of transformations: function inlining, constant folding, and normalization. The normalization is necessary to bring constants next to each other in the AST such that we can fold them using pattern matching. An example of such a normalization can be seen in Fig. 6.11.

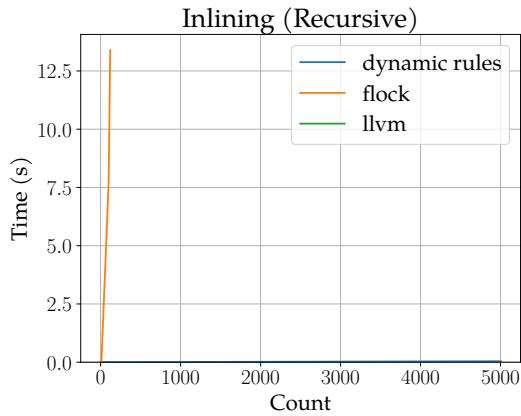


Figure 6.9: Run time of optimization pipelines on various sizes of the program shown in Fig. 6.10

```

let function a0(n: int): int = n + 1
  function a1(n1: int): int = a0(n1) + 1
  function a2(n2: int): int = a1(n2) + 1
in
    a2(1)
end

```

Figure 6.10: Example program for $n=3$

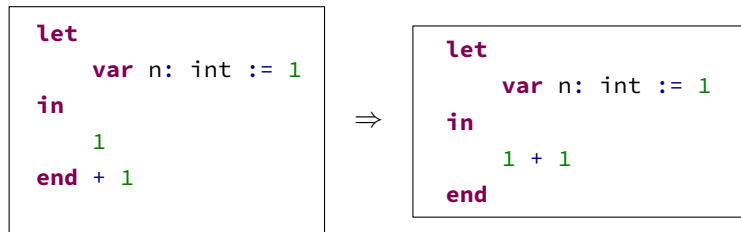


Figure 6.11: Normalization transformation that is applied before a peephole optimization can be used to fold the addition

This transformation is communicated to the Flock runtime through the `flock-replace-node` strategy as a replacement of the entire `let` term with a new `let` term. This lack of granularity causes Flock to recompute the control-flow graph and connected components for the entire term, even though the declarations in the `let` are unchanged. In our benchmark program this inner term grows linearly with the number of functions inlined, as each inlining introduces a new `let` term. This also means our optimizer must repeatedly apply the normalization transformation, to move the addition further inside the `let` structure. Due to the large cost of each individual transformation this causes a rapid slowdown as the input program grows. A single additional function requires n normalization transformations. We discuss a potential solution to this problem in the Section 8.1.

6.2 Memory Analysis of an Optimization Pipeline

6.2.1 Benchmarking Methodology

We measure the memory usage of the Flock and Dynamic Rules optimizers during the optimization process for each of the previously discussed programs where $n = 5000$. We include the results for only two of the benchmarks because the other benchmarks showed near identical results. For Flock, we measure the total amount of memory used whenever a value analysis query was performed. Similarly for DR, we measure the total amount of memory used whenever a dynamic rule was invoked to query the value of a reference. In both instances we first request the JVM to run the garbage collector to attempt to reduce noise caused by garbage memory. We do not measure the memory usage of LLVM for two reasons. First, the compiled nature of LLVM makes it difficult to compare results to the Flock and DR projects even though we aim to reduce the noise caused by garbage collection. Second, the complexity of the LLVM codebase makes it difficult to implement a comparable method of memory management.

6.2.2 Results

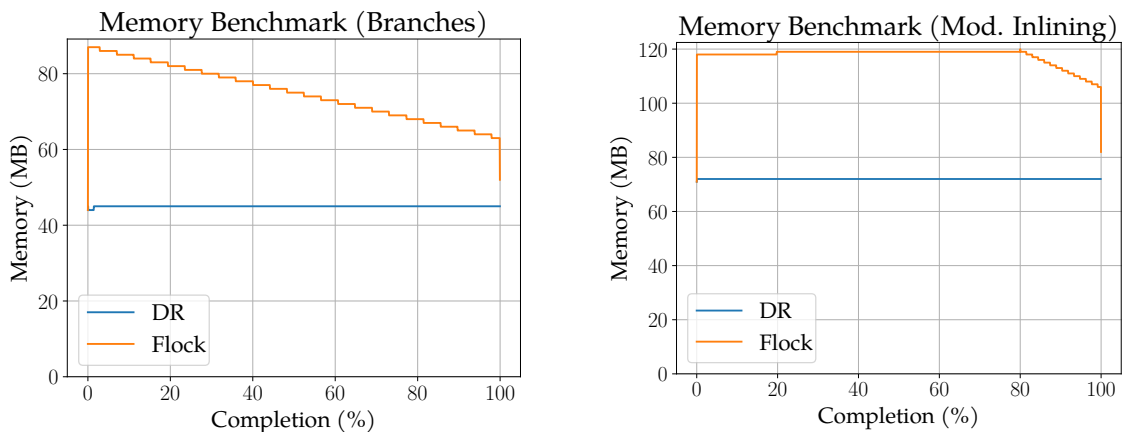


Figure 6.12: Memory usage of Flock and Dynamic Rules in various benchmarks. Vertically the line represents the memory usage of the optimizer during the optimization of a program, horizontally the line represents a measurement as a percentage of total measurements. The graphs do not depict runtime.

Fig. 6.12 shows the memory usage of Flock and Dynamic Rules during the optimization of the two benchmarks. We see a similar pattern in both benchmarks. The memory usage of Flock sharply rises as the runtime is initialized when the control-flow graph and connected components are created. When the size of the program decreases due to an optimization so does the memory usage decrease. Dynamic Rules does not create these structures, and we thus do not see the same increase and decrease in memory usage. The small increases in memory usage can be attributed to the creation of dynamic rules.

6.3 Analysis of Query Latencies

6.3.1 Benchmarking Methodology

To measure query latencies we perform 5000 value analysis queries on the *Vars* benchmark with $n = 5000$. For each variable reference in the program we query the value analysis results for that reference. The variable references of the program are processed in random order to show the effects of caching in the Flock runtime. We measure the latency for each query and sort them in descending order. We include five warmup runs for both implementations to reduce the noise caused by JVM warmup.

6.3.2 Results

Fig. 6.13 shows the latencies for 5000 value queries sorted in descending order. The Dynamic Rules implementation does not include caching, and we can see that the latency for a query correlates linearly to how far a variable reference is from the root of the program. The Flock implementation does support caching (with no additional effort from the programmer), and we see that there are few queries with high latencies ($> 10\text{ms}$), and many queries with very low latencies ($< 0.1\text{ms}$). We find that Flock outperforms Dynamic Rules in use cases where no program transformations are performed and initialization is performed in advanced, such as when a programmer uses dataflow analysis to inspect the behaviour of a program.

6.4 Threats to Validity

In this section we discuss some threats to the validity of the results shown and discussed in this chapter. In summary, we find that the strength of these results are limited by three aspects:

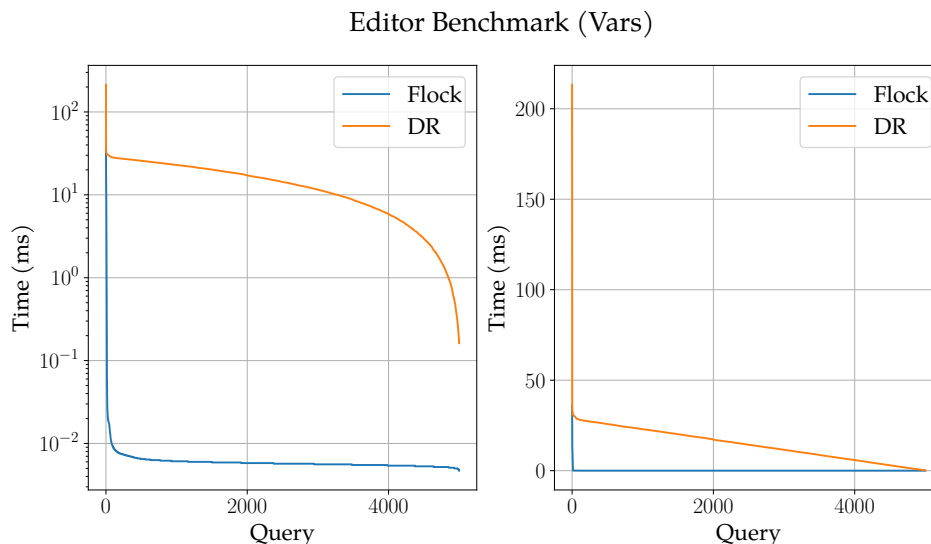


Figure 6.13: Sorted latencies for value analysis queries shown for Flock (cached) and Dynamic Rules (uncached) on a logarithmic (left) and linear (right) scale

- The synthetic nature of the benchmarks
- The small set of analyses and optimizations implemented
- The lack of complex control-flow constructs in Tiger

6.4.1 Synthetic Benchmarks

Our quantitative benchmarking consists of generating programs with predictable structure to stress-test specific aspects of each implementation. While these programs are far from realistic, they still give insight into the behaviour of the performance of Flock.

Ideally we would also benchmark real programs, as this would give us more confidence in the significance of the results, however we do not have a database of programs written in Tiger which are large enough to draw reliable conclusions from. Synthetic benchmarks are thus a best effort to gain useful insight into the performance of Flock. It has provided us with several unexpected results, leading to a better understanding of the runtime and improvements to the implementation. In Ch. 8 we discuss future work to improve this aspect of the research.

6.4.2 Small Set of Analyses

The set of optimizations and analyses included in the Flock pipeline is only a small subset of the total number of dataflow analyses used in an industrial compiler such as LLVM. We have implemented value, liveness, and very busy expression analyses for use in optimizations, as well as an array length analysis for use in static analysis. The limited number of analyses implemented and tested presents a threat to the validity of this research as the performance characteristics measured in this chapter may differ when a larger number of analyses is executed.

6.4.3 Lack of Complex Control-Flow Constructs

Not all types of control-flow are (fully) supported by Flock. First, constructs that require information about names in the program such as `goto` are not supported, because there is no access to name information when writing a FlowSpec specification using Flock. Second are constructs that do not require name information, but require non-local control-flow. Examples of this are `return`, and `throw` statements. These are supported by Flock, but their strongly

connected components cannot be efficiently updated. We discuss this in more detail as part of the future work in Ch. 8.

Chapter 7

Related work

In this chapter we will discuss work related to incremental program analysis and optimization. We subdivide the relevant related work into three areas: dataflow analysis, incremental dataflow analysis, and incremental Datalog.

7.1 Dataflow Analysis

Dataflow analysis was introduced by Gary Kildall as a tool for implementing optimizations in compilers (Kildall 1973). The technique has become a cornerstone in the field of program optimization, and is included in many educational texts on the topic in its own chapter such as the *The Dragon Books* (Aho, Lam, et al. 1986; Aho, Ullman, et al. 1977) and *Engineer a Compiler* (Cooper and Torczon 2011).

Dataflow analysis can be found in many industrial compiler projects such as Clang/LLVM (C. Lattner and Adve 2004; *Clang* 2022), GCC (Hayes 1999), the OCaml compiler (*OCaml Source Code* 2022), WebKit (*WebKit Source Code* 2022), etc, where it is used to implement analyses that are used for optimization. Another use case for dataflow analysis is in program checking to ensure that programs are valid, such as in Java where it is used to ensure the initialization of `final` fields (Gosling et al. 2000).

FlowSpec (Smits and Visser 2017) is a DSL within the Spoofox language workbench (Kats and Visser 2010) for declarative dataflow analysis specification. FlowSpec aims to provide language developers with a concise and expressive tool for implementing dataflow analyses, as an improvement over operational encodings. We discussed the limitations of FlowSpec in Ch. 3. This thesis reimplements the FlowSpec language as Flock with support for incremental execution of analyses.

Similar to FlowSpec, Dynamic Rules (Bravenboer et al. 2006) is a technique of implementing dataflow analysis within Spoofox. In contrast to FlowSpec this approach does not introduce a DSL for dataflow analysis, instead adding the necessary language features in Stratego directly. The limitations of Dynamic Rules are similarly discussed in Ch. 3.

Dataflow analysis has also been used in integrated development environments (IDEs) to provide additional functionality to programmers. The JetBrains IDE contains functionality that lets a programmer invoke a dataflow analysis and inspect the results to better understand a program (*Jetbrains Dataflow Analysis* 2022).

7.2 Incremental Dataflow Analysis

There is a large body of work relating to incremental dataflow analysis algorithms from the 80's. Incremental dataflow analysis algorithms as defined in Ryder (1983) take a dataflow solution and a program change, and aim to efficiently and correctly update the dataflow solution. This is a different problem than we are trying to solve, as we do not assume a full dataflow solution is present during program changes, nor do we require a full dataflow solution to be computed afterwards. Nevertheless most work from this period aims to solve this problem (Ryder 1983; Zadeck 1984; Carroll and Ryder 1988; Marlowe and Ryder 1989; Pollock and Soffa 1989). It is not clear to us if these approaches can be adapted to our problem.

FlowSpec also supports user-defined lattices, whereas these algorithms commonly make assumptions about the abstract domain. Finally, there is a lack of empirical evidence on the performance of these algorithms.

Demand-driven analysis is an approach that aims to compute only the necessary dataflow information to answer a given query. Several demand-driven analysis algorithms have been developed (Horwitz, Reps, and Sagiv 1995; Heintze and Tardieu 2001; Sridharan et al. 2005; Lu et al. 2013). These approaches map a dataflow analysis problem to a path reachability problem. Instead of propagating information from the root of the program to each program point, we can start at the point for which information is queried, and seek a path from the program point back to the root through which the fact may be realized. This avoids unnecessary work to answer a query. By caching analysis results future queries may be computed more efficiently. This approach is more granular than our forward-propagation approach but requires that the size of dataflow facts is finite, and that the dataflow functions are distributive. Its application to interprocedural analysis is also different from our intraprocedural domain.

Lerner et al. (Lerner, Grove, and Chambers 2002) present a framework for program analysis and optimization that can automatically compose analyses. Their solution solves the phase-ordering problem while avoiding manually implementing a superanalysis. The results show that automatically composed analyses generate similarly efficient code compared to a manual superanalysis, with a compile-time overhead of less than 20%. Their framework does not support non-local graph replacements, and does not show the size or complexity of the implementation of the analyses aside from their composability. Further work focusses on soundness of transformations (Lerner, Millstein, and Chambers 2003; Lerner, Millstein, Rice, et al. 2005; Kundu, Tatlock, and Lerner 2009), which is relevant to this thesis but outside of its scope.

7.3 Incremental Datalog

Datalog is a declarative logic programming language that has found use in a number of areas, including program analysis. By representing program structure (such as AST and CFG information) and analyses (such as the dataflow rules) as relations, Datalog can be used to solve and query analysis results. Existing work on incrementality for Datalog has also been successfully applied to program analysis.

The DRed algorithm (Gupta, Mumick, and Subrahmanian 1993) is an algorithm for incrementally maintaining relational and deductive databases, and supports negation, aggregation, and recursion. A limitation of the DRed algorithm is that it does not support user-defined recursive aggregation, which limits its application to analyses with abstract domains of powersets.

Based on the DRed algorithm, IncA (Szabó, Erdweg, and Voelter 2016) is a domain specific language for the definition of incremental program analysis. Szabo et al. introduce *DRed_L* (Szabó, Bergmann, et al. 2018), an extension of DRed, which supports aggregation over custom lattices. At a higher level IncA aims to provide language/IDE developers with the tools to implement efficient incremental analyses that can be used for optimization, IDE features, etc. Results show that analyses written in IncA can scale well to large programs with low latency (measured in milliseconds). While powerful and flexible, the relational paradigm of IncA is further removed from the domain of dataflow analysis than FlowSpec. Fig. 7.1 shows the definition of control-flow rules for an if-statement in compared to the equivalent FlowSpec definition. 7.2, which gives an impression of the difference in complexities of the specifications written in these two languages.

```

def cIf(trg : Statement): Statement = {
  src := precedingStatement(trg)
  assert src instanceof IfStatement
  return lastStatement(src)
} alt {
  src := precedingStatement(trg)
  assert src instanceof IfStatement
  assert undef src.else
  return src
} alt {
  assert undef precedingStatement(trg)
  parent := trg.parent
  assert parent instanceof IfStatement
  return parent
}

```

Figure 7.1: Control-flow graph rules for a C-style if in IncA (Szabó, Erdweg, and Voelter 2016)

```

If(c, t, e) = entry -> c -> t -> exit,
              c -> e -> exit

```

Figure 7.2: Control-flow graph rules for a C-style if in FlowSpec

Chapter 8

Conclusion

This thesis presents an implementation and runtime called Flock for the FlowSpec DSL. Our aim was to answer the following research questions:

- **RQ1** What does a flexible but efficient dataflow analysis and optimization framework for language developers look like within Spooifax?
- **RQ2** How can we efficiently and automatically compose dataflow analyses and optimizations without imposing strong restrictions on language developers?
- **RQ3** How does the efficiency of such a framework compare to other approaches when applied to optimization and analysis?
- **RQ4** How much effort is required to implement an optimization pipeline, static analysis, and editor services with such a framework?

RQ1 In Chapter 4 we showed what Flock looks like when applied to an optimization pipeline and editor services for Tiger. We find that the complexity of the Stratego code that queries the Flock runtime is low, and the FlowSpec specifications can remain unchanged compared to the original implementation. By exposing a Stratego API that returns the analysis results as Stratego terms, the queries can be easily used to perform transformations. Using hooks exposed by Spooifax in Stratego we can easily provide editor services.

RQ2 Chapter 5 contains a description of the implementation of Flock. We presented the internal datastructures used by the runtime and the analyses, and how we can incrementally maintain them when program transformations are performed. Flock does not impose restrictions on the type of transformations that can be performed, the direction or composition of analyses, the abstract domain of the analyses, etc. Flock does not support all the necessary types of complex control-flow, which is an area of future work.

RQ3 The experimental evaluation is discussed in Chapter 6. For the purpose of optimization, we find that while Flock is significantly slower than LLVM and DR in some cases, it sometimes outperforms the other solutions. One large outlier is due to the lack of granularity in the Spooifax API. Overall the performance of Flock is likely good enough for DSLs or language prototypes, and there are several areas of future work that may improve its performance and allow for broader applicability. When applied to editor services, we find that the query latency of Flock is lower than that of the DR approach, especially when caching takes effect.

RQ4 Chapter 4 contains the exposition of control-flow semantics, dataflow semantics, optimizations, and editor services implemented using Flock in Spooifax. We find that the FlowSpec specifications remain unchanged from the existing (non-incremental) implementation, and that the Stratego code used to invoke the runtime is succinct, but we are not able to draw strong conclusions from this due to a lack of user studies.

```
let
  function nfactor(n: int): int =
    if n = 0 then 1 else (n * nfactor(n-1))
in
  3
end + 1
```

Figure 8.1: An optimizable program that does not match a peephole optimization

```
let
  function nfactor(n: int): int =
    if n = 0 then 1 else (n * nfactor(n-1))
in
  3 + 1
end
```

Figure 8.2: An optimizable program that will match a peephole optimization

8.1 Future Work

There is a broad array of possible directions in which to extend this research, some of which we will discuss in this section.

Granularity of API Not all transformations can be efficiently represented with the API exposed by Flock. A common transformation that we cannot express efficiently occurs when we are normalizing an AST. Consider a peephole optimization that folds arithmetic expressions, and the program shown in Fig. 8.1. The peephole optimizer will not be able to optimize this program as the addition is not in the expected form. As a possible solution we can *normalize* the program, for example by pushing all arithmetic expressions to the leafs of the AST. The result of this normalization is shown in Fig. 8.2. To express this transformation in terms of the Flock API we must communicate a replacement of the entire `let`-term, including the unchanged `nfactor` function. This leads the runtime to unnecessarily recompute the CFG and SCCs for this function. The effects of this are clearly visible in the benchmark shown in Section 6.1.6. We would like to be able to express that some subtrees are unchanged, such that the runtime can reuse their derived CFGs and SCCs.

Phase-Ordering Problem The phase-ordering problem is a problem related to the ordering of optimization passes within an optimizing compiler. The ordering of passes can have a strong influence on the effectiveness of the optimizer. An example of existing work that tackles the phase ordering problem is Lerner et al. (Lerner, Grove, and Chambers 2002). A possible avenue of future work is to similarly implement a type of eager optimization that can transform the program before a fixpoint is reached. It has been shown that this approach can unlock optimizations that cannot be achieved by executing optimization passes separately.

User Testing We were not able to draw strong conclusions on the ease of use of the Flock framework due to a lack of users. An interesting area of future work will be to find users for the Flock framework and gather information about the perceived ease of use and complexity of the framework. A possible avenue for this is to integrate the use of Flock in the Compiler Construction course at the TU Delft, in which students may use the framework to implement optimizations for their language.

Complex Control-Flow As discussed in Ch. 6, the lack of support for name-dependent control-flow such as `goto`, and the fallback support for control-flow such as `return` present a threat to the validity of this research. This limitation has implications for the applicability of the current implementation of Flock as many languages make use of such control-flow constructs. The challenge in supporting these efficiently comes from the difficulty in incrementally computing the strongly connected components of a control-flow graph with complex control flow. One avenue may be found in using general algorithms for maintaining the SCCs of our CFG. The problem is known as *fully dynamic connectivity* in literature (Roditty and Zwick 2016; Holm, Lichtenberg, and Thorup 2001). It is not clear what the impact on performance will be if implemented. For our Tiger prototype, the control-flow rules fit within the current limitations, since the language does not contain these control-flow constructs.

We hypothesize that the performance impact of the lack of efficient incremental updates for such constructs depends mostly on the type of transformations that are performed. The

impact is negligible if the program terms that have special control-flow semantics are never part of the transformations. This is true for optimizations such as constant propagation and constant folding, but false for others such as inlining, loop merging, etc.

Bibliography

- Aho, Alfred V, Monica S Lam, et al. (1986). *Compilers: principles, techniques and tools*.
- Aho, Alfred V, Jeffrey D Ullman, et al. (1977). *Principles of compiler design*. Addison-Wesley Pub. Co.
- Appel, Andrew W (2004). *Modern compiler implementation in C*. Cambridge university press.
- Bravenboer, Martin et al. (2006). "Program transformation with scoped dynamic rewrite rules". In: *Fundamenta Informaticae* 69.1-2, pp. 123–178.
- Carroll, Martin D and Barbara G Ryder (1988). "Incremental data flow analysis via dominator and attribute update". In: *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 274–284.
- Clang (2022). <https://clang.llvm.org/>. [Online; accessed 16-June-2022].
- Clang Dataflow (2022). <https://clang.llvm.org/docs/DataFlowSanitizer.html>. [Online; accessed 28-March-2022].
- Cooper, Keith D and Linda Torczon (2011). *Engineering a compiler*. Elsevier.
- Dataflow Analysis (2022). <https://rustc-dev-guide.rust-lang.org/mir/dataflow.html>. [Online, accessed 28-March-2022].
- Ellson, John et al. (2001). "Graphviz—open source graph drawing tools". In: *International Symposium on Graph Drawing*. Springer, pp. 483–484.
- Gosling, James et al. (2000). *The Java language specification*. Addison-Wesley Professional.
- Gupta, Ashish, Inderpal Singh Mumick, and Venkatramanan Siva Subrahmanian (1993). "Maintaining views incrementally". In: *ACM SIGMOD Record* 22.2, pp. 157–166.
- Hayes, Michael (1999). *GCC Source Code - gcc/df-core.c*.
- Heintze, Nevin and Olivier Tardieu (2001). "Demand-driven pointer analysis". In: *ACM SIGPLAN Notices* 36.5, pp. 24–34.
- Holm, Jacob, Kristian de Lichtenberg, and Mikkel Thorup (July 2001). "Poly-Logarithmic Deterministic Fully-Dynamic Algorithms for Connectivity, Minimum Spanning Tree, 2-Edge, and Biconnectivity". In: *J. ACM* 48.4, pp. 723–760. ISSN: 0004-5411. DOI: 10.1145/502090.502095. URL: <https://doi.org/10.1145/502090.502095>.
- Hong, Sungpack et al. (2012). "Green-Marl: a DSL for easy and efficient graph analysis". In: *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, pp. 349–362.
- Horwitz, Susan, Thomas Reps, and Mooly Sagiv (1995). "Demand interprocedural dataflow analysis". In: *ACM SIGSOFT Software Engineering Notes* 20.4, pp. 104–115.
- Jetbrains Dataflow Analysis (2022). <https://www.jetbrains.com/help/idea/analyzing-data-flow.html>. [Online; accessed 22-March-2022].
- JMH: Java Microbenchmark Harness (2022). <https://openjdk.org/projects/code-tools/jmh/>. [Online; accessed 15-June-2022].

- Kats, Lennart CL and Eelco Visser (2010). “The Spoofox language workbench: rules for declarative specification of languages and IDEs”. In: *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pp. 444–463.
- Kildall, Gary A (1973). “A unified approach to global program optimization”. In: *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pp. 194–206.
- Kundu, Sudipta, Zachary Tatlock, and Sorin Lerner (2009). “Proving optimizations correct using parameterized program equivalence”. In: *ACM Sigplan Notices* 44.6, pp. 327–337.
- Lattner, Chris and Vikram Adve (2004). “LLVM: A compilation framework for lifelong program analysis & transformation”. In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004. IEEE*, pp. 75–86.
- Lattner, Chris Arthur (2002). “LLVM: An infrastructure for multi-stage optimization”. PhD thesis. University of Illinois at Urbana-Champaign.
- Lerner, Sorin, David Grove, and Craig Chambers (2002). “Composing dataflow analyses and transformations”. In: *ACM SIGPLAN Notices* 37.1, pp. 270–282.
- Lerner, Sorin, Todd Millstein, and Craig Chambers (2003). “Automatically proving the correctness of compiler optimizations”. In: *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pp. 220–231.
- Lerner, Sorin, Todd Millstein, Erika Rice, et al. (2005). “Automated soundness proofs for dataflow analyses and transformations via local rules”. In: *ACM SIGPLAN Notices* 40.1, pp. 364–377.
- Leroy, Xavier et al. (2016). “CompCert—a formally verified optimizing compiler”. In: *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*.
- Lu, Yi et al. (2013). “An incremental points-to analysis with CFL-reachability”. In: *International Conference on Compiler Construction*. Springer, pp. 61–81.
- Marlowe, Thomas J and Barbara G Ryder (1989). “An efficient hybrid algorithm for incremental data flow analysis”. In: *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 184–196.
- OCaml Source Code (2022). <https://github.com/ocaml/ocaml/blob/trunk/asmcomp/dataflow.mli>. [Online; accessed 16-June-2022].
- Pizlo, Filip (2016). *Introducing the B3 JIT Compiler*. <https://webkit.org/blog/5852/introducing-the-b3-jit-compiler/>. [Online; accessed 29-March-2022].
- Pollock, Lori L and Mary Lou Soffa (1989). “An incremental version of iterative data flow analysis”. In: *IEEE Transactions on Software Engineering* 15.12, pp. 1537–1549.
- Pornin, Thomas (n.d.). *Why Constant-Time Crypto?* <https://www.bearssl.org/constanttime.html>. [Online; accessed 25-March-2022].
- Roditty, Liam and Uri Zwick (2016). “A Fully Dynamic Reachability Algorithm for Directed Graphs with an Almost Linear Update Time”. In: *SIAM J. Comput.* 45, pp. 712–733.
- Ryder, Barbara G (1983). “Incremental data flow analysis”. In: *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pp. 167–176.
- Smits, Jeff and Eelco Visser (2017). “FlowSpec: declarative dataflow analysis specification”. In: *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, pp. 221–231.
- Souza Amorim, Luís Eduardo de and Eelco Visser (2020). “Multi-Purpose Syntax Definition with SDF3”. In: *International Conference on Software Engineering and Formal Methods*. Springer, pp. 1–23.
- Spencer, Henrey (1997). URL: <https://compilers.iecc.com/comparch/article/97-10-017>.
- Sridharan, Manu et al. (2005). “Demand-driven points-to analysis for Java”. In: *ACM SIGPLAN Notices* 40.10, pp. 59–76.
- Steindorfer, Michael Johannes (2017). “Efficient immutable collections”. In.

- Szabó, Tamás, Gábor Bergmann, et al. (2018). “Incrementalizing lattice-based program analyses in Datalog”. In: *Proceedings of the ACM on Programming Languages* 2.OOPSLA, pp. 1–29.
- Szabó, Tamás, Sebastian Erdweg, and Markus Voelter (2016). “Inca: A dsl for the definition of incremental program analyses”. In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pp. 320–331.
- Tarjan, Robert (1972). “Depth-first search and linear graph algorithms”. In: *SIAM journal on computing* 1.2, pp. 146–160.
- Visser, Eelco (2004). “Program transformation with Stratego/XT”. In: *Domain-specific program generation*. Springer, pp. 216–238.
- WebKit Source Code (2022). <https://trac.webkit.org/browser/trunk/Source/JavaScriptCore/dfg/DFGObjectAllocationSinkingPhase.cpp>. [Online; accessed 16-June-2022].
- Yang, Xuejun et al. (2011). “Finding and understanding bugs in C compilers”. In: *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pp. 283–294.
- Zadeck, Frank Kenneth (1984). “Incremental data flow analysis in a structured program editor”. In: *Proceedings of the 1984 SIGPLAN symposium on Compiler construction*, pp. 132–143.

Appendix A

Tiger FlowSpec Control-Flow Spec

The following is the full Tiger control-flow specification written in FlowSpec.

```
module tiger/Cfg
signature
sorts
  Id = string
  StrConst = string
  IntConst = int
  Exp
  LoopBinding
  Dec
  Type
  Occ
  TypeId
  FArg
  Var
  Int
  LValue

constructors
  Mod : Exp -> Exp
  ProcDec : Occ * list(FArg) * Exp -> Dec
  FunDec : Occ * list(FArg) * Type * Exp -> Dec
  VarDec : Id * Type * Exp -> Dec
  VarDecNoType : Occ * Exp -> Dec
  TypeDec : Occ * Type -> Dec
  Type : TypeId -> Type
  Occ : Id -> Occ
  Tid : Occ -> TypeId

  UMinus : Exp -> Exp
  Minus : Exp * Exp -> Exp
  Plus : Exp * Exp -> Exp
  Times : Exp * Exp -> Exp
  Divide : Exp * Exp -> Exp
```

Lt : Exp * Exp -> Exp
Gt : Exp * Exp -> Exp
Eq : Exp * Exp -> Exp
Geq : Exp * Exp -> Exp
Leq : Exp * Exp -> Exp
Neq : Exp * Exp -> Exp
And : Exp * Exp -> Exp
Or : Exp * Exp -> Exp

Call : Occ * **list**(Exp) -> Exp
If : Exp * Exp * Exp -> Exp
IfThen : Exp * Exp -> Exp
Seq : **list**(Exp) -> Exp
For : LoopBinding * Exp -> Exp
LoopBinding : Var * Exp * Exp -> LoopBinding
Assign : LValue * Exp -> Exp
Let : **list**(Dec) * **list**(Exp) -> Exp
Return : Exp

String : StrConst -> Exp
Int : IntConst -> Exp
Var : Id -> Var
Array : TypeId * Exp * Exp -> Exp
Subscript : LValue * Exp -> LValue
Exp : LValue -> Exp
LValue : Var -> LValue

control-flow rules

root Mod(s) = **start** -> s -> **end**
root ProcDec(n, args, body) = **start** -> body -> **end**
root FunDec(n, args, rt, body) = **start** -> body -> **end**

ProcDec(_, _, _) = **entry** -> **exit**
FunDec(_, _, _, _) = **entry** -> **exit**
TypeDec(_, _) = **entry** -> **exit**

TypeDec(occ, t) = **entry** -> **exit**
VarDec(n, t, e) = **entry** -> e -> **this** -> **exit**
VarDecNoType(n, e) = **entry** -> e -> **this** -> **exit**

UMinus(exp) = **entry** -> exp -> **this** -> **exit**
Minus(lhs, rhs) = **entry** -> lhs -> rhs -> **this** -> **exit**
Plus(lhs, rhs) = **entry** -> lhs -> rhs -> **this** -> **exit**
Times(lhs, rhs) = **entry** -> lhs -> rhs -> **this** -> **exit**
Divide(lhs, rhs) = **entry** -> lhs -> rhs -> **this** -> **exit**
Lt(lhs, rhs) = **entry** -> lhs -> rhs -> **this** -> **exit**
Gt(lhs, rhs) = **entry** -> lhs -> rhs -> **this** -> **exit**
Eq(lhs, rhs) = **entry** -> lhs -> rhs -> **this** -> **exit**
Geq(lhs, rhs) = **entry** -> lhs -> rhs -> **this** -> **exit**

```
Leq(lhs, rhs) = entry -> lhs -> rhs -> this -> exit
Neq(lhs, rhs) = entry -> lhs -> rhs -> this -> exit
And(lhs, rhs) = entry -> lhs -> rhs -> this -> exit
Or(lhs, rhs) = entry -> lhs -> rhs -> this -> exit
Subscript(lval, idx) = entry -> idx -> lval -> this -> exit

Call(_, args) = entry -> args -> this -> exit
If(c, t, e) = entry -> c -> t -> exit,
             c -> e -> exit
LValue(inner) = entry -> inner -> exit
IfThen(c, t) = entry -> c -> t -> exit
Assign(lval, expr) = entry -> expr -> lval -> this -> exit
Seq(stmts) = entry -> stmts -> exit
For(binding@LoopBinding(var, from, to), body) =
    entry -> from -> to -> binding -> body -> binding,
    binding -> exit
LoopBinding(var, from, to) = entry -> this -> exit
Let(decs, exps) = entry -> decs -> exps -> exit
Array(_, len, init) = entry -> len -> init -> this -> exit
Return() = entry -> end

node Var(_)
node Int(_)
node String(_)
```


Appendix B

Tiger FlowSpec Data-Flow Spec

The following is the Tiger value analysis data-flow specification written in FlowSpec.

```
module tiger/ValueAnalysis

imports
  tiger/Cfg

properties
  values: SimpleMap[string, Value]

property rules
  values(_.end) = {}
  values(prev -> VarDec(n, _, Int(i)))
    = { k |-> v | (k |-> v) <- values(prev), k != n } \\/ {n |-> Const(i)}
  values(prev -> VarDec(n, _, _))
    = { k |-> v | (k |-> v) <- values(prev), k != n } \\/ {n |-> Top()}
  values(prev -> Assign(LValue(Var(n)), Int(i)))
    = { k |-> v | (k |-> v) <- values(prev), k != n } \\/ {n |-> Const(i)}
  values(prev -> Assign(LValue(Var(n)), _))
    = { k |-> v | (k |-> v) <- values(prev), k != n } \\/ {n |-> Top()}
  values(prev -> LoopBinding(Var(n), _, _))
    = { k |-> v | (k |-> v) <- values(prev), k != n } \\/ {n |-> Top()}
  values(prev -> _) = values(prev)

types
  ConstProp =
  | Top()
  | Const(int)
  | Bottom()

lattices
  Value where
    type = ConstProp
    bottom = Bottom()
    top = Top()
```

The following is the Tiger liveness analysis data-flow specification written in FlowSpec.

```
module tiger/LiveVariableAnalysis
imports
  tiger/Cfg

properties
  live: MaySet[string]

property rules
  live(_.end) = {}
  live(VarDec(n, _, _) -> next) =
    {m | m <- live(next), m != n}
  live(Assign(LValue(Var(n)), _) -> next) =
    {m | m <- live(next), m != n}
  live(LoopBinding(Var(n), _, _) -> next) =
    {m | m <- live(next), m != n}
  live(Var(n) -> next) = {n} \ / live(next)
  live(_ -> next) = live(next)
```

The following is the Tiger array length analysis data-flow specification written in FlowSpec.

```
module tiger/ArrayLengthAnalysis

imports
  tiger/Cfg

properties
  lengths: SimpleMap[string, Value]

property rules
  lengths(_.end) = SimpleMap[string, Value].bottom

  lengths(prev -> VarDec(n, _, Array(_, Int(i), _))) =
    { k |-> v | (k |-> v) <- lengths(prev), k != n } \ / {n |-> Const(i)}

  lengths(prev -> VarDec(n, _, _)) =
    { k |-> v | (k |-> v) <- lengths(prev), k != n } \ / {n |-> Top()}

  lengths(prev -> Assign(LValue(Var(n)), Array(_, Int(i), _))) =
    { k |-> v | (k |-> v) <- lengths(prev), k != n } \ / {n |-> Const(i)}

  lengths(prev -> Assign(LValue(Var(n)), _)) =
    { k |-> v | (k |-> v) <- lengths(prev), k != n } \ / {n |-> Top()}

  lengths(prev -> LoopBinding(Var(n), _, _)) =
    { k |-> v | (k |-> v) <- lengths(prev), k != n } \ / {n |-> Top()}

  lengths(prev -> _) = lengths(prev)

types
  Length =
  | Top()
  | Const(int)
  | Bottom()
```