Implementing a Functional Language for Flix

by

Ming-Ho Yee

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2016

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Static program analysis is a powerful technique for maintaining software, with applications such as compiler optimizations, code refactoring, and bug finding. Static analyzers are typically implemented in general-purpose programming languages, such as C++ and Java; however, these analyzers are complex and often difficult to understand and maintain. An alternate approach is to use Datalog, a declarative language. Implementors can express analysis constraints declaratively, which makes it easier to understand and ensure correctness of the analysis. Furthermore, the declarative nature of the analysis allows multiple, independent analyses to be easily combined.

Flix is a programming language for static analysis, consisting of a logic language and a functional language. The logic language is inspired by Datalog, but supports user-defined lattices. The functional language allows implementors to write functions, something which is not supported in Datalog. These two extensions, user-defined lattices and functions, allow Flix to support analyses that cannot be expressed by Datalog, such as a constant propagation analysis. Datalog is limited to constraints on relations, and although it can simulate finite lattices, it cannot express lattices over an infinite domain. Finally, another advantage of Flix is that it supports interoperability with existing tools written in general-purpose programming languages.

This thesis discusses the implementation of the Flix functional language, which involves abstract syntax tree transformations, an interpreter back-end, and a code generator back-end. The implementation must support a number of interesting language features, such as pattern matching, first-class functions, and interoperability.

The thesis also evaluates the implementation, comparing the interpreter and code generator back-ends in terms of correctness and performance. The performance benchmarks include purely functional programs (such as an N-body simulation), programs that involve both the logic and functional languages (such as matrix multiplication), and a real-world static analysis (the Strong Update analysis). Additionally, for the purely functional benchmarks, the performance of Flix is compared to C++, Java, Scala, and Ruby.

In general, the performance of compiled Flix code is significantly faster than interpreted Flix code. This applies to all the purely functional benchmarks, as well as benchmarks that spend most of the time in the functional language, rather than the logic language. Furthermore, for purely functional code, the performance of compiled Flix is often comparable to Java and Scala.

Acknowledgements

I would like to thank everyone who has supported me throughout this journey:

First, my supervisor, Ondřej Lhoták, for the time and effort he put into advising me. His answers to my questions were always meaningful, insightful, and helpful, and he always found the time to meet with me, even when busy or on sabbatical.

Werner Dietl and Gregor Richards, my committee members, whose comments have greatly improved my thesis.

Magnus Madsen, friend and mentor over the past year, who has taught me much about graduate student life.

My labmates and friends, Dan Brotherston and Marianna Rapoport, for our technical discussions, and also non-technical discussions when we needed a break from work.

My friends from the local quizbowl club and larger community.

And finally, my family. My mother and father, Wai-Sin Yee and Siu-Pok Yee, and my sister, Ming-Cee Yee, for always being there for me.

This research was supported by funding from the Natural Sciences and Engineering Research Council of Canada, the Government of Ontario, the University of Waterloo, and Professor David R. Cheriton.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

As software becomes increasingly widespread and complex, it becomes more important to manage its complexity and correctness. One powerful technique for this task is static program analysis, with applications such as compiler optimizations, code refactoring, and bug finding. A static analysis typically models abstract program state as elements of lattices, where the lattice ordering represents precision. Elements lower in the lattice correspond to more precise information about the program under analysis. The statements of the program constrain the solution to the analysis problem, and the goal is to find the most precise—and therefore minimal—solution. This solution can be computed by iterating from the bottom element of the lattice, until reaching a fixed point that satisfies the constraints [18, 45].

Static analyzers, which are commonly implemented in general-purpose programming languages such as C++ and Java, are themselves complex pieces of software and can be difficult to maintain. Furthermore, they involve many specific implementation details, such as the choice of evaluation strategy and data structures, which become tightly coupled with the analysis. The details are necessary for solving the fixed point problem of the analysis; however, it becomes more challenging to understand and ensure correctness of the actual analysis. Another problem is that optimizations must be manually implemented for each static analyzer and cannot easily be reused.

An alternate approach is to implement static analyses in Datalog, a declarative programming language. Datalog has its roots in Prolog, logic programming, and deductive databases [16, 36]. A Datalog program is a set of constraints over relations, where each relation is a set of facts and is therefore an element of the power set lattice of facts. The solution to a Datalog program is the minimal model that satisfies those constraints, and is computed by a Datalog solver. One useful property of Datalog is that every program

is guaranteed to terminate with the minimal model. A programmer can clearly write the constraints of a problem in Datalog, leave the specifics of computing the solution to the solver, and be assured that the analysis will terminate. Additionally, an independent solver allows the programmer to choose between different solver implementations, and an optimization only needs to be implemented once per solver. These advantages make Datalog a useful language for implementing static analyses, such as points-to analyses of Java programs [3, 13, 71, 80]. The declarative nature of the analysis makes it easier to understand and maintain, and also easier to combine with other independent analyses.

However, Datalog has limitations that restrict its expressiveness. Many important analyses, such as constant propagation, are based on more general lattices than just relations, and cannot be represented in Datalog. Furthermore, Datalog does not support user-defined functions. In some cases, these limitations can be worked around by embedding a lattice in a power set, or by explicitly tabulating inputs and outputs to simulate a function. However, these approaches are cumbersome and inefficient, and impossible for lattices and functions with infinite domains. Furthermore, most Datalog implementations have poor interoperability with existing tools—typically, a front-end tool extracts input facts from the program under analysis, and then provides the facts to a Datalog solver in a textual format.

Flix is a new programming language designed to address these limitations [53]. The implementation is open source and available on GitHub[1]. The semantics of Flix are based on Datalog and preserve Datalog's termination guarantee. However, Flix extends Datalog to support user-defined lattices and functions, allowing a larger class of analyses to be expressed. Constraints on lattices and relations are expressed in a logic language with Datalog-like syntax, while functions are expressed in a pure functional language with Scala-like syntax. Furthermore, since Flix is implemented on the Java Virtual Machine (JVM), it easily supports interoperability with JVM languages.

The main contributions of this thesis are the implementation of the functional language of Flix and its evaluation. The implementation involves abstract syntax tree transformations, an interpreter back-end, and a code generator back-end, while the evaluation compares both back-ends in terms of correctness and performance.

The thesis is structured as follows: Chapter 2 describes the background for this thesis, specifically an informal overview of Flix, its architecture, and its interoperability with the JVM. Chapter 3 examines the implementation of the functional language. Chapter 4 discusses the evaluation in terms of correctness and performance, comparing both back-ends and also benchmarks implemented in other languages. Chapter 5 explores related work, Chapter 6 considers future work, and Chapter 7 concludes.

[1] https://github.com/flix/flix

Chapter 2

Background

This chapter discusses the background material necessary for understanding the thesis. It provides an informal overview of the Flix language (2.1), the architecture of the Flix implementation (2.2), and a brief description of interoperability with Flix (2.3). The discussion in this chapter is not meant to be a detailed tutorial or a formal specification.

2.1 The Flix Language

Flix is a programming language for static analysis. It is based on Datalog, but extended with user-defined lattices and functions. Flix consists of a logic language with Datalog-like syntax and a functional language with Scala-like syntax. This section first walks through a simple Flix implementation of a constant propagation analysis (2.1.1), before discussing the logic language (2.1.2) and the functional language (2.1.3). Two examples illustrate the semantics of the Flix language.

2.1.1 Constant Propagation in Flix

The example in Figure 2.1 expresses a constant propagation analysis, but with some details simplified. Line 1 defines the `ConstProp` namespace. Since all definitions in this example are within the `ConstProp` namespace, function and constant names do not need to be fully qualified. Lines 2–4 declare the `Constant` tagged union, whose members, or tags, represent elements of the constant propagation lattice. When referenced, a tag must be prefixed by its enum name. Lines 6–12 define the `leq` function, where the function body is an expression

```
namespace ConstProp {
  enum Constant {
    case Top, case Cst(Int), case Bot
  }

  def leq(e1: Constant, e2: Constant): Bool =
    match (e1, e2) with {
      case (Constant.Bot, _)                    => true
      case (Constant.Cst(n1), Constant.Cst(n2)) => n1 == n2
      case (_, Constant.Top)                    => true
      case _                                    => false
    }

  // definitions omitted for brevity
  def lub(e1: Constant, e2: Constant): Constant = ...
  def glb(e1: Constant, e2: Constant): Constant = ...

  let Constant<> = (Constant.Bot, Constant.Top, leq, lub, glb)

  // definitions omitted for brevity
  def sum(e1: Constant, e2: Constant): Constant = ...
  def isMaybeZero(e: Constant): Bool = ...

  // analysis inputs
  rel AsnStm(r: Str, c: Int)         // r = c
  rel AddStm(r: Str, x: Str, y: Str) // r = x + y
  rel DivStm(r: Str, x: Str, y: Str) // r = x / y

  // analysis outputs
  lat LocalVar(k: Str, v: Constant)
  rel ArithError(r: Str)

  LocalVar(r, Constant.Cst(c)) :- AsnStm(r, c).
  LocalVar(r, sum(v1, v2)) :- AddStm(r, x, y),
                              LocalVar(x, v1),
                              LocalVar(y, v2).
  ArithError(r) :- isMaybeZero(v),
                   DivExp(r, x, y),
                   LocalVar(y, v).
}
```

Figure 2.1: An example Flix program implementing a simple constant propagation analysis.

written in the Flix functional language. For brevity, the function bodies of `lub` and `glb` are omitted. Line 18 enables lattice semantics for the `Constant` tagged union, specifying its top element (`Constant.Top`), its bottom element (`Constant.Bot`), the partial order on the lattice (`leq`), the least upper bound operator (`lub`), and the greatest lowest bound operator (`glb`).

Line 21 declares the `sum` *transfer function* and line 22 declares the `isMaybeZero` *filter function*. Their function bodies are omitted, again for brevity. Lines 25–27 define the `AsnStm`, `AddStm`, and `DivStm` relations, and represent the analysis inputs. Lines 30–31 define the `LocalVar` lattice and `ArithError` relation, and represent the analysis outputs. The `LocalVar` lattice is similar to a relation declaration, but its last attribute must be a lattice type, in this case, `Constant`. Finally, lines 33–39 define the constraints of the analysis, which are written as rules in the Flix logic language.

2.1.2 The Logic Language

Constraints on lattices and relations are expressed as rules in the Flix logic language. Like Datalog, a *rule* in Flix consists of a *head* on the left-hand side of `:-` (read as "if" or "is implied by"), and an optional *body* on the right-hand side. A rule without a body is called a *fact*. The head is an *atom* while the body is a list of atoms. In Flix, the body may also include a list of filter function applications. An atom consists of a *predicate symbol* with *terms* as its arguments. A term is either a constant value or a variable. In Flix, the last term of the head predicate may contain transfer function applications. Figure 2.2 summarizes the syntax of Flix rules.

The semantics of the Flix logic language are based on the semantics of Datalog: if the body of a rule is satisfied by some set of existing facts, then the head of the rule must also be satisfied. This infers a new fact, which may cause rules to be re-evaluated and more facts to be inferred. This process continues until no more new facts are inferred—in other words, when the fixed point is reached. The resulting set of facts is called the solution, or minimal model.

Flix extends Datalog in three ways, by supporting lattices, filter functions, and transfer functions. Furthermore, as long as every lattice has finite height and every function is strict and monotone, a Flix program is guaranteed to terminate with the minimal model. The formal semantics are described in the original Flix paper [53], but two examples are provided here as an informal overview.

$$
\begin{array}{lll}
\langle \textit{rule} \rangle & ::= \langle \textit{head} \rangle \texttt{ :- } \langle \textit{body} \rangle \texttt{ .} & \text{(rule)} \\
& \mid \langle \textit{head} \rangle \texttt{ .} & \text{(fact)} \\
\langle \textit{head} \rangle & ::= \langle \textit{atom} \rangle & \text{(head)} \\
\langle \textit{body} \rangle & ::= \langle \textit{fun-apps} \rangle \texttt{ , } \langle \textit{atoms} \rangle & \text{(body)} \\
& \mid \langle \textit{atoms} \rangle & \\
\langle \textit{atom} \rangle & ::= \text{pred } \texttt{(} \langle \textit{terms} \rangle \texttt{)} & \text{(atom)} \\
\langle \textit{atoms} \rangle & ::= \langle \textit{atoms} \rangle \texttt{ , } \langle \textit{atom} \rangle & \\
& \mid \langle \textit{atom} \rangle & \\
\langle \textit{fun-app} \rangle & ::= \text{ident } \texttt{(} \langle \textit{terms-opt} \rangle \texttt{)} & \text{(function application)} \\
\langle \textit{fun-apps} \rangle & ::= \langle \textit{fun-apps} \rangle \texttt{ , } \langle \textit{fun-app} \rangle & \\
& \mid \langle \textit{fun-app} \rangle & \\
\langle \textit{term} \rangle & ::= \text{val} \mid \text{var} \mid \langle \textit{fun-app} \rangle & \text{(term)} \\
\langle \textit{terms} \rangle & ::= \langle \textit{terms} \rangle \texttt{ , } \langle \textit{term} \rangle & \\
& \mid \langle \textit{term} \rangle & \\
\langle \textit{terms-opt} \rangle & ::= \langle \textit{terms} \rangle \mid \epsilon &
\end{array}
$$

Figure 2.2: The grammar of Flix rules. Only the last term of the head predicate may contain function applications, but for simplicity, this restriction is not enforced by the grammar.

Example 1

Consider the constant propagation analysis in Figure 2.1. To keep this example simple, only the first rule of the analysis will be examined:

```
LocalVar(r, Constant.Cst(c)) :- AsnStm(r, c).
```

The rule states that if the value `c` is assigned to the variable `r`, then the local variable `r` has the abstract value `Constant.Cst(c)`.

The analysis will run on two input facts, which have been extracted from the program under analysis by a separate front-end tool:

```
AsnStm("x", 0).
AsnStm("x", 1).
```

These facts represent a variable `x` that has been assigned the values `0` and `1`. It is unknown whether the assignment occurred in the same or different control flow branches. The two facts cause the above rule to be evaluated, which infers two new facts:

```
LocalVar("x", Constant.Cst(0)).
LocalVar("x", Constant.Cst(1)).
```

Under Datalog semantics, `LocalVar` would be a relation, and the two inferred facts above would be the solution to the analysis. However, this is a Flix program where `LocalVar` has been declared as a *lattice*, so the semantics—and thus the solution—is different.

The Flix semantics treat `LocalVar` as a map lattice, where all the attributes except for the last represent the key, while the last attribute represents the value. Since the two facts have the same key (`"x"`) but different values (`Constant.Cst(0)` and `Constant.Cst(1)`), Flix merges the values by taking the least upper bound. This is implemented by calling the `lub` function with the two values, which returns `Constant.Top`. Therefore, the solution is a single fact:

```
LocalVar("x", Constant.Top).
```

The static analysis was unable to compute an exact value for `x`. However, `x` has some value, which Flix approximates as `Constant.Top`.

Example 2

The previous example illustrated the basic idea behind Flix lattices and how they differ from Datalog relations. This second example will reinforce those differences, and also demonstrate transfer functions and filter functions.

As before, consider the constant propagation analysis, with the program under analysis encoded as the following input facts:

```
1 AsnStm("a", 0).         // a = 0
2 AsnStm("b", 0).         // b = 0
3 AsnStm("c", 1).         // c = 1
4                         // if (mystery())
5 AddStm("d", "a", "b"). //   d = a + b
6                         // else
7 AddStm("d", "a", "c"). //   d = a + c
8                         //
9 DivStm("e", "c", "d"). // e = c / d
```

The first three `AsnStm` facts cause evaluation of the rule

```
LocalVar(r, Constant.Cst(c)) :- AsnStm(r, c).
```

This generates the three following facts:

```
LocalVar("a", Constant.Cst(0)).
LocalVar("b", Constant.Cst(0)).
LocalVar("c", Constant.Cst(1)).
```

These new facts, along with the `AddStm` input facts on lines 5 and 7, cause evaluation of the second rule

```
LocalVar(r, sum(v1, v2)) :- AddStm(r, x, y),
                            LocalVar(x, v1),
                            LocalVar(y, v2).
```

The rule states that if `r` is assigned the value `x + y`, the variable `x` has value `v1`, and the variable `y` has value `v2`, then the local variable `r` has the value `sum(v1, v2)`, where `sum` is a transfer function that must be evaluated. This generates two facts with the same key (`"d"`) but different values (`Constant.Cst(0)` and `Constant.Cst(1)`), which—due to the Flix lattice semantics—are merged by calling the `lub` function, to generate a single fact:

```
LocalVar("d", Constant.Top).
```

This new fact for `d`, along with the input fact on line 9, causes evaluation of the rule

```
ArithError(r) :- isMaybeZero(v),
                 DivExp(r, x, y),
                 LocalVar(y, v).
```

The rule states that if `r` is assigned the value `x / y`, the variable `y` has the value `v`, and if the function application `isMaybeZero(v)` evaluates to `true`, then there may be an arithmetic error involving `r`. This infers the fact

```
ArithError("e").
```

If `isMaybeZero(v)` evaluated to `false`, then the rule would not infer a new fact.

At this point, no further evaluation of the rules is necessary and the minimal model has been computed:

```
LocalVar("a", Constant.Cst(0)).
LocalVar("b", Constant.Cst(0)).
LocalVar("c", Constant.Cst(1)).
LocalVar("d", Constant.Top).
ArithError("e").
```

The analysis has computed exact values for the variables `a`, `b`, and `c`, but not for `d`, because of a branch in the control flow of the program. To maintain soundness, the analysis returned the approximate result, `Constant.Top`. Finally, the analysis has identified a possible arithmetic error involving the variable `e`.

2.1.3 The Functional Language

Expressions in Flix are implemented in a pure and strict functional language, with Scala-like syntax. Figure 2.3 summarizes the grammar of expressions. An expression can be a literal, variable reference, function call, built-in unary or binary expression, let-binding, if-expression, pattern match expression, tag expression, or tuple expression.

Flix supports a wide variety of literals and data types, including unit, booleans, characters, floating point numbers, integers, and strings. Floating point numbers can be single- or double-precision. Integers are signed and can have 8, 16, 32, 64, or arbitrarily many bits. Flix also supports two compound types: tuples and tagged unions. A member of a tagged union, or a tag, must be written as `enum.tag val`, where `enum` is the enum name (`Constant` in the Figure 2.1 example), `tag` is the member (`Top`, `Cst`, or `Bot`), and `val` is the inner tag value (an integer for `Constant.Cst`), which may be omitted if it is unit.

⟨*exp*⟩	::=	lit	(literal)
	|	ident	(variable)
	|	⟨*fqn*⟩ `(` ⟨*exps-opt*⟩ `)`	(call)
	|	⟨*unop*⟩ ⟨*exp*⟩	(unary)
	|	⟨*exp*⟩ ⟨*binop*⟩ ⟨*exp*⟩	(binary)
	|	`let` ident `=` ⟨*exp*⟩ `in` ⟨*exp*⟩	(let)
	|	`if (` ⟨*exp*⟩ `)` ⟨*exp*⟩ `else` ⟨*exp*⟩	(if)
	|	`match` ⟨*exp*⟩ `with {` ⟨*cases*⟩ `}`	(match)
	|	ident `.` ident `(` ⟨*exps-opt*⟩ `)` | ident `.` ident	(tag)
	|	`(` ⟨*exps*⟩ `,` ⟨*exp*⟩ `)`	(tuple)
⟨*exps*⟩	::=	⟨*exps*⟩ `,` ⟨*exp*⟩ | ⟨*exp*⟩	
⟨*exps-opt*⟩	::=	⟨*exps*⟩ | ϵ	
⟨*fqn*⟩	::=	⟨*fqn-prefix*⟩ `/` ident	
⟨*fqn-prefix*⟩	::=	⟨*fqn-prefix*⟩ `.` ident | ident	
⟨*unop*⟩	::=	`!` | `+` | `-` | `~`	
⟨*binop*⟩	::=	`+` | `-` | `*` | `/` | `%` | `**`	(arithmetic)
	|	`<` | `<=` | `>` | `>=` | `==` | `!=`	(comparison)
	|	`&&` | `||` | `==>` | `<==>`	(logical)
	|	`&` | `|` | `^` | `<<` | `>>`	(bitwise)
⟨*case*⟩	::=	`case` ⟨*pattern*⟩ `=>` ⟨*exp*⟩	
⟨*cases*⟩	::=	⟨*cases*⟩ ⟨*case*⟩	
⟨*pattern*⟩	::=	`_`	(wildcard)
	|	ident	(variable)
	|	lit	(literal)
	|	ident `.` ident `(` ⟨*patterns-opt*⟩ `)` | ident `.` ident	(tag)
	|	`(` ⟨*patterns*⟩ `,` ⟨*pattern*⟩ `)`	(tuple)
⟨*patterns*⟩	::=	⟨*patterns*⟩ `,` ⟨*pattern*⟩ | ⟨*pattern*⟩	
⟨*patterns-opt*⟩	::=	⟨*patterns*⟩ | ϵ	

Figure 2.3: The grammar of Flix expressions.

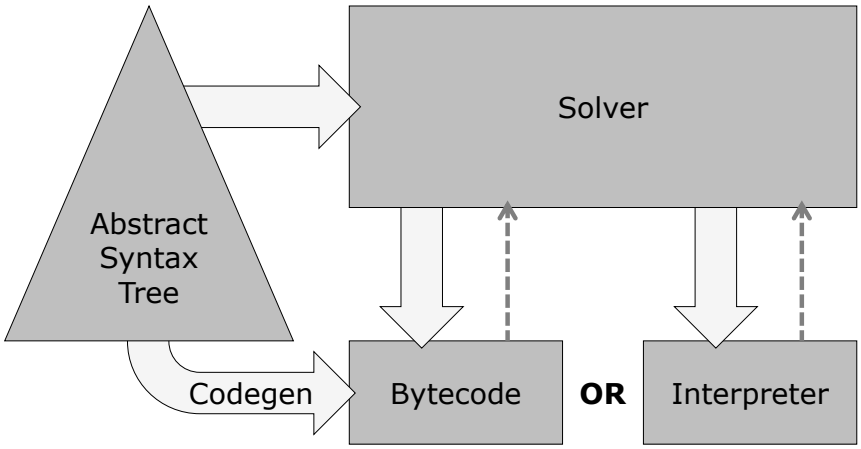

Figure 2.4: Architecture of the Flix back-end. The `ExecutableAst` is the input to the back-end, which consists of the logic language implementation (solver) and the functional language implementation (code generator or interpreter).

2.2 Architecture of the Flix Implementation

Flix is implemented in Scala [61], a functional and object-oriented programming language that runs on the Java Virtual Machine (JVM) [51]. This allows Flix to run on any platform with a JVM, and furthermore, easily integrate with existing JVM tools.

The front-end architecture of Flix is typical for a compiler, with phases for parsing, weeding, name resolution, and type checking. Each phase produces an abstract syntax tree (AST), which is fed into the next phase. These ASTs are the `ParsedAst`, `WeededAst`, `ResolvedAst`, and `TypedAst`. The result of the type checking phase, the `TypedAst`, undergoes a number of transformations before reaching the back-end. These transformations include compiling higher-level constructs to lower-level primitives, closure conversion and lambda lifting, and variable numbering. The process creates a `SimplifiedAst` and then an `ExecutableAst`, which serves as the input to the back-end.

The architecture of the back-end is depicted in Figure 2.4. It has two components: the logic language implementation and the functional language implementation. The logic language is implemented by the solver, which evaluates rules in the program. The solver was developed by another member of the research team and is outside the scope of this thesis—a paper concerning the solver is currently in submission. The functional language has two implementations: an AST interpreter and a JVM bytecode generator. The interpreter was the original back-end, which is easier to understand and maintain, while the code generator is the default back-end, as it provides better performance.

Execution in the back-end starts in the solver, which evaluates the rules in the logic language using a fixed-point algorithm. As the solver computes a solution, it may need to evaluate expressions written in the functional language. To do so, the solver calls the generated bytecode or the interpreter. This happens in two cases: when the solver needs to evaluate a lattice operation, such as `lub`, or when a filter or transfer function application explicitly appears in a rule. Once the functional code has been evaluated, the resulting value is returned to the solver, which then continues its computation.

2.3 Interoperability with Flix

One of the goals of Flix is to integrate with existing tools and front-ends, without having to communicate through a textual interface. Flix can be called as a JVM library, and returns solutions as JVM objects. Furthermore, the application programming interface (API) allows Flix code to call external JVM functions, or hooks.

A hook must be implemented as a function object that implements the functional interface `Invokable`. This interface is provided by Flix and declares a single abstract method `apply`. The method takes an array of `IValue`s, representing the function arguments, and returns an `IValue`. Hooks are programmatically added to Flix by calling the `addHook` API method, which takes a name, the Flix type of the hook, and the object implementing the external function. Within Flix, the name is bound to the hook and can be called as if it were a Flix function.

The `IValue` interface abstracts and hides the internal implementation details of Flix types and values. This allows the implementation to change without affecting the interface or breaking existing code. However, there is a performance penalty, as all Flix values need to be wrapped and unwrapped as `IValue` objects. Users who wish to avoid this penalty—at the risk of having the implementation change—can use the `InvokableUnsafe` interface and the `addHookUnsafe` method. This interface avoids wrapping, but all Flix values are cast to `java.lang.Object`. The user must then manually cast `java.lang.Object` to the actual type of the Flix value.

This interoperability feature allows static analysis implementors to write Flix functions in languages such as Java or Scala, which offers more flexibility than using just the Flix functional language. An implementor could also integrate Flix into an existing framework such as Soot [78] or WALA [25]—the framework would extract facts from the program under analysis, call Flix as a library, receive the solution from Flix, and then continue with further analysis. Both of these use cases are difficult, if not impossible, with existing Datalog implementations.

Chapter 3

Implementation

This chapter comprises the main contribution of the thesis, discussing the implementation of the functional language of Flix. It discusses the abstract syntax tree (AST) transformations (3.1), the interpreter back-end (3.2), and the code generator back-end (3.3).

Although the code examples in this chapter resemble Flix, they are not valid Flix programs. The pseudocode omits details, contains features not currently supported by Flix, and includes constructs that have no source-level representation.

3.1 Abstract Syntax Tree Transformations

The front-end phases perform parsing, weeding, name resolution, and type checking, producing a `TypedAst` that directly corresponds to the source program. However, before the AST can be consumed by the back-end, further transformations are required. A phase transforms the `TypedAst` into the `SimplifiedAst`, compiling pattern matching into lower-level primitives (3.1.1), and further phases modify the `SimplifiedAst` directly, by performing closure conversion and lambda lifting (3.1.2), as well as variable numbering (3.1.3).

Currently, the Flix implementation does not include any optimization phases, but they could be applied at this point. Once optimizations (if any) are complete, the AST enters the final phase, which creates an `ExecutableAst`. This AST is nearly identical to the `SimplifiedAst`, except that lists are converted to arrays for performance, and mutable fields are added to record performance and debugging information. The interpreter back-end directly interprets the `ExecutableAst`, while the code generator back-end consumes the `ExecutableAst` to produce JVM bytecode.

```
// before
match x with {
  case PAT1 => EXP1
  case PAT2 => EXP2
  case _    => ERROR // implicit default case
}

// after
let v_0 = x in
let err = λ() ERROR in
let e_2 = λ() if (PAT2 succeeds) EXP2 else err() in
let e_1 = λ() if (PAT1 succeeds) EXP1 else e_2() in
  e_1()
```

Figure 3.1: An example pattern match expression before and after transformation, shown in pseudocode. The pattern match expression is compiled into a series of nested let-bindings and lambda functions.

3.1.1 Compiling Pattern Match Expressions

The functional language of Flix supports pattern matching: expressions can be matched against wildcards, bound to variables, or matched against literals, and tag and tuple expressions can be destructed. Some languages allow pattern matching in parameters of function definitions, but Flix only supports pattern matching in case expressions. The full grammar for pattern match expressions is described in Figure 2.3 and an example is shown in the `leq` function of Figure 2.1.

Pattern matching in the source program directly corresponds to nodes in the `TypedAst`. The match expression in the upper half of Figure 3.1 corresponds to a `Match` node, with a child node for the expression to be matched (e.g. `x`) and a list of cases. Each case contains a pattern node (e.g. `PAT1`) and a body expression node (e.g. `EXP1`). The interpreter could consume the `TypedAst` and directly implement pattern matching; however, this is less straightforward for the code generator. For consistency, both the interpreter and code generator use the `ExecutableAst`, rather than having the interpreter use the `TypedAst` and the code generator use the `ExecutableAst`.

The `TypedAst` representation of pattern matching must be transformed into lower-level primitives in the `SimplifiedAst`, as shown in the lower half of Figure 3.1. (Recall that the `SimplifiedAst` is transformed into an `ExecutableAst` before being consumed by the back-end.) The `SimplifiedAst` primitives include let-bindings, if-expressions, and lambda

functions, as well as primitives without corresponding source-level syntax: `CheckTag`, `GetTagValue`, and `GetTupleIndex`. `CheckTag(t,v)` checks the value `v` against the tag name `t`, `GetTagValue(v)` extracts the inner value of a tag `v`, and `GetTupleIndex(v,i)` extracts the element at the index `i` from the tuple value `v`.

Pattern-matching compilation [11] is implemented as the functions `Expression.simplify` and `Pattern.simplify`. `Expression.simplify` rewrites a match expression into a series of nested let-bindings and lambda functions, as shown in Figure 3.1. On the other hand, `Pattern.simplify` converts a single pattern, for instance `PAT1`, into a lambda body. Each lambda function represents a single case which calls the next function if the match fails.

`Expression.simplify` starts by generating fresh variable names for the expression to be matched, each of the cases, and the implicit default case. The expression to be matched is then bound to one of these fresh variables. In Figure 3.1, this binds `x` to `v_0`, allowing `x` to be evaluated only once while `v_0` can be referenced multiple times. Next, `Expression.simplify` binds an error function to a name, for example `err`. The error function represents the body of the implicit default case, which throws an error if none of the other cases successfully match at run time.

The remaining cases are handled in reverse order. This builds a hierarchy of nested let-bindings from outside-in, making the names of each case visible to the preceding cases. For instance, `Expression.simplify` starts by processing the last non-default case, calling `Pattern.simplify` to convert `PAT2`, and then binding a lambda function to the fresh variable `e_2`. At run time, if the match succeeds, then `EXP2` is evaluated; otherwise the next case, `err`, is called. If the cases were processed in forward order, then `err` would not be visible from within `e_2`. Finally, the body of the innermost let-binding contains a call to the first case, `e_1`, which begins evaluation of the pattern match expression.

`Pattern.simplify` is called by `Expression.simplify` and converts each individual pattern, rather than the entire pattern match expression. `Pattern.simplify` is a recursive function taking four arguments: a list of patterns, a list of variable names, a "success" expression, and a "fail" expression. Initially, each list contains a single element, representing the original pattern and variable name. However, additional patterns and temporary variable names are prepended to the lists whenever a subpattern is processed.

In the base case for `Pattern.simplify`, the pattern and variable lists are empty, so the function generates code to evaluate the "success" expression. In the recursive case, `Pattern.simplify` generates code to match the first variable against the first pattern, and then recurses on the lists. At run time, a successful match will evaluate the next variable and pattern, while a failure will evaluate the "fail" expression. The details for each of the different patterns are discussed in the subsections below.

Wildcard Pattern

A wildcard pattern match always succeeds. The example in Figure 3.2 matches `x` against a wildcard, so it always returns the body, `true`. This translates to the function `e_1` whose body is simply the body of the case. No equality checks or let-bindings are required.

```
// before
match x with {
  case _ => true
}

// after
let v_0 = x in
let err = λ() ERROR in
let e_1 = λ() true in
  e_1()
```

Figure 3.2: A wildcard pattern match before and after transformation.

Variable Pattern

Similar to the wildcard pattern, a variable pattern match always succeeds. However, a variable pattern also binds the matched value to the variable name in the pattern. This is typically used in a subpattern to extract values from tags and tuples. In Figure 3.3, the pattern match binds `x` to the name `n`, and the case body adds `1` to `n`. This translates to a let-binding in the compiled pattern match.

```
// before
match x with {
  case n => n + 1
}

// after
let v_0 = x in
let err = λ() ERROR in
let e_1 = λ() (let n = v_0 in n + 1) in
  e_1()
```

Figure 3.3: A variable pattern match before and after transformation.

Literal Pattern

A literal pattern match only succeeds if the expression to be matched equals the literal in the pattern. In Figure 3.4, the pattern match compares `x` to `42`. If it succeeds, the expression returns `true`. Otherwise, the default case is evaluated and an error is thrown. This translates to an if-expression, where the condition is the equality comparison, the true branch is the case body, and the false branch calls the next case.

```
// before
match x with {
  case 42 => true
}

// after
let v_0 = x in
let err = λ() ERROR in
let e_1 = λ() if (v_0 == 42)
                true
              else
                err() in
  e_1()
```

Figure 3.4: A literal pattern match before and after transformation.

Tag Pattern

A tag pattern match succeeds if the expression to be matched has the same enum and tag names as the tag in the pattern, and the subpattern successfully matches. The example in Figure 3.5 defines a tagged union `E` with the tags `A` and `B`, which respectively contain an integer value and a string value. Here, `E` is the enum name, while `A` and `B` are the tag names. The pattern match checks that `x` is an `E.A` tag, and if so, matches the subpattern against the inner value of the tag. In this example, the subpattern is a literal pattern that compares the tag value to `42`.

The pattern match translates to an if-expression, where the condition uses the `CheckTag` primitive. Note that the type checker ensures `x` is a member of the tagged union `E`, so `CheckTag` only needs to check that `x` is an `A` tag. The true branch of the if-expression first extracts the tag value with `GetTagValue` and binds it to the fresh variable name `n_0`. Then the translated subpattern compares `n_0` to `42`. If the subpattern succeeds, the case body `true` is returned. If either check fails, the next case is called.

```
enum E {
  case A(Int),
  case B(Str)
}

// before
match x with {
  case E.A(42) => true
}

// after
let v_0 = x in
let err = λ() ERROR in
let e_1 = λ() if (CheckTag(A, v_0))
                let n_0 = GetTagValue(v_0) in
                  if (n_0 == 42)
                    true
                  else
                    err()
              else
                err() in
  e_1()
```

Figure 3.5: A tag pattern match before and after transformation.

Tuple Pattern

While a tag pattern contains a single subpattern, a tuple pattern contains multiple subpatterns, each corresponding to an element of the tuple. The type checker guarantees that the arity and type of the tuple to be matched equals the arity and type of the tuple pattern. Therefore, all the pattern match needs to do is bind each of the tuple elements to fresh variable names, and then match those variables against the subpatterns.

In Figure 3.6, `x` is guaranteed to be a tuple of two integers. The tuple pattern translates to a series of let-bindings, where the tuple elements are extracted with `GetTupleIndex` and bound to the fresh variable names `n_0` and `n_1`. Next, each of the subpatterns is translated. In this example, they are literal patterns, and compare the first tuple element `n_0` to `4` and the second tuple element `n_1` to `2`. If both succeed, the case body `true` is returned; otherwise, the next case is called.

```
// before
match x with {
  case (4, 2) => true
}

// after
let v_0 = x in
let err = λ() ERROR in
let e_1 = λ() let n_0 = GetTupleIndex(v_0, 0) in
              let n_1 = GetTupleIndex(v_0, 1) in
                if (n_0 == 4)
                  if (n_1 == 2)
                    true
                  else
                    err()
                else
                  err() in
  e_1()
```

Figure 3.6: A tuple pattern match before and after transformation.

3.1.2 Closure Conversion and Lambda Lifting

At this time, Flix has partial support for lambda functions. Specifically, in the back-end, lambda functions are completely implemented, since they are required for pattern matching. However, the front-end does not yet support lambda functions, so users cannot write lambda expressions in the source program.

Similar to pattern matching, lambda functions could be directly implemented in the interpreter. However, this is not possible for code generation, since all functions are compiled to bytecode methods, which must be defined at the top-level and cannot be nested. Furthermore, a function call must apply its arguments to some function reference—it cannot apply arguments to an arbitrary expression that evaluates to a function.

Lambda Lifting

The first problem, only allowing top-level function definitions, could be solved by a transformation called *lambda lifting* [42]. Consider the example program below:

```
def f() = let g = λ(x, y) x+y in
            g(1, 2)
```

The nested definition for g is not allowed in bytecode, so the transformation lifts it:

```
def f() = g(1, 2)
def g(x, y) = x+y
```

This also binds the function to a name, allowing the code generator to implement calls to named functions. However, the approach fails if the nested function contains free variables:

```
def f(a) = let g = λ(x, y) a+x+y in
             g(1, 2)
```

Within g, the variables x and y are bound and their values are provided by the arguments to the lambda function. However, a is a free variable and has no binding once g is lifted:

```
def f(a) = g(1, 2)
def g(x, y) = a+x+y  // what is a?
```

Therefore, lambda lifting must modify the lambda function's parameter list by prepending the free variables:

```
def f(a) = let g = λ(a′, x, y) a′+x+y in
             g(a, 1, 2)
```

Now g can be lifted:

```
def f(a) = g(a, 1, 2)
def g(a′, x, y) = a′+x+y
```

Note that the transformation also rewrote the call to g, since it now takes three arguments.

Closure Conversion

Unfortunately, lambda lifting does not work for higher-order functions: if g is returned from a function or passed as an argument to a function, it becomes more difficult to identify the call sites that need to be rewritten. It may not even be possible to supply the value for the newly added parameter:

```
def f(a) = let g = λ(a′, x, y) a′+x+y in
             h(g, 1, 2)
def h(g′, x, y) = g′(x, y) // how to rewrite g′?
```

The actual problem is that the variables in the lambda function need to be bound at two distinct times. `x` and `y` are the original parameters and are bound as arguments when the function is *called*. However, `a` is bound when the lambda function is *defined*. The solution is to convert the lambda function into a closure, a structure that contains code (the rewritten lambda function) and data (the captured values that need to be passed to the lambda function). These captured values are bound when the closure is created, and stored so they can be applied when the closure is called. This transformation is called *closure conversion* [8].

In the `SimplifiedAst`, a closure is created by a `MkClosure` node and called by an `ApplyClosure` node:

```
def f(a) = let g = MkClosure(λ(a′, x, y) a′+x+y, a) in
           h(g, 1, 2)
def h(g′, x, y) = ApplyClosure(g′, x, y)
```

When the `MkClosure` node is evaluated, it saves `a` as the value for `a′`. Later, when the `ApplyClosure` node is evaluated, the saved value `a` and the arguments `1` and `2` are provided to the closure function.

In the Flix implementation, closure conversion occurs after pattern matching has been compiled. The closure conversion phase traverses the `SimplifiedAst`, replacing `Lambda` nodes with `MkClosure` nodes and calls to lambda functions with `ApplyClosure` nodes. Lambda bodies are recursively converted, since Flix allows nested lambda definitions.

Closure conversion solves the problems with free variables and higher-order functions, but lambda functions are still defined within the closure, rather than at the top-level. However, it is now safe to apply lambda lifting:

```
def f(a) = let g = MkClosure(f′, a) in
           h(g, 1, 2)
def h(g′, x, y) = ApplyClosure(g′, x, y)
def f′(a′, x, y) = a′+x+y
```

The lambda lifting phase traverses the `SimplifiedAst` while maintaining a map of names to lambda functions. As each `Lambda` node is encountered, nested lambda functions are recursively lifted. Then the phase creates a fresh name for the lambda function, updates the map, and replaces the `Lambda` node with a function reference. After the entire AST has been traversed, the list of top-level definitions is updated with the definitions in the map.

Higher-Order Functions and Hooks

The closure conversion phase makes two other replacements to support higher-order functions and hooks. Consider the following example, where `g` is higher-order function that calls its first argument with its second argument, but is given a reference to a top-level function:

```
def f(x) = x + 1
def g(x, y) = ApplyClosure(x, y)
def h() = g(f, 1) // f needs to be a closure
```

In the function body of `g`, `x` is an expression and not a function reference; therefore, it is called as a closure. However, in the arguments to `g`, `f` is a function reference and needs to be replaced by a `MkClosure` node:

```
def f(x) = x + 1
def g(x, y) = ApplyClosure(x, y)
def h() = g(MkClosure(f), 1)
```

The second replacement occurs when a hook reference is being passed to a higher-order function. Consider the following example, where `f` is an external function that takes a single argument:

```
def g(x, y) = ApplyClosure(x, y)
def h() = g(f, 1) // f needs to be a closure
```

This is similar to the first case, except that `f` is a hook reference rather than function reference. `MkClosure` can only take a lambda function or a function reference, so the closure conversion phase wraps the hook reference inside a lambda function:

```
def g(x, y) = ApplyClosure(x, y)
def h() = g(λ(z) f(z), 1)
```

Now the newly created lambda function is closure converted

```
def g(x, y) = ApplyClosure(x, y)
def h() = g(MkClosure(λ(z) f(z)), 1)
```

and then lifted:

```
def g(x, y) = ApplyClosure(x, y)
def h() = g(MkClosure(h′, 1)
def h′(z) = f(z)
```

These two replacements complete the implementation of higher-order functions in Flix, without any additional effort from the user programmer.

3.1.3 Variable Numbering

At run time, each JVM method stores its local variables in the local variable array, with indices starting from zero. The JVM provides bytecode instructions for storing variables from the operand stack and loading variables onto the operand stack. Each entry in the variable array stores a 32-bit value, and to store 64-bit values, the JVM uses two adjacent entries in the array. For instance, a `long` referenced by the index n is stored in the entries indexed by n and $n + 1$. The JVM also uses the local variable array to pass arguments: starting from zero, entries in the local variable array are initialized to argument values. Subsequent entries are then used for local variables.

Variable numbering is the final phase applied to the `SimplifiedAst`, before constructing the `ExecutableAst`. This phase assigns indices to local variables in the AST, so the code generator can use these indices to emit variable stores and loads. The interpreter ignores these indices, since it references variables by their names.

The variable numbering phase has a straightforward implementation. It traverses the AST, maintaining a map of variable names to indices, as well as a counter for the next index to use. The traversal assigns indices to arguments first, from left to right, and then local variables (defined in let-bindings) in order of appearance. The map is updated and the counter incremented for each variable definition. For values of type `long` and `double`, which require two entries in the array, the counter is incremented twice. When a variable is referenced, the appropriate index is looked up from the map.

The type checking phase guarantees that each variable is defined before use. However, variable shadowing is permitted. Consider the following, where the subscripts represent indices assigned by the variable numbering phase, and are not part of the source program:

```
1 let x0 = 10 in
2   let x1 = x0 + 15 in
3     x1 // value is 25
```

The example contains two definitions for the variable `x`, one in each let-binding. On line 2, $\texttt{x}_0$ is bound in the outer let-binding on line 1 and has the value `10`. Evaluating $\texttt{x}_0$ `+ 15` returns `25`, which is bound to the new definition $\texttt{x}_1$, and hides the previous definition $\texttt{x}_0$. Thus, on line 3, $\texttt{x}_1$ has value `25`.

Flix does not allow mutation or sequencing, so once a variable has been shadowed, it can never be referenced again. Therefore, the variable numbering phase can naïvely overwrite indices in its map. In the above example, the phase first assigns the index `0` to `x`, but upon reaching the definition on line 2, it reassigns the index `1` to `x`.

3.2 Interpreting the AST

The interpreter back-end for Flix consumes an `ExecutableAst`. Much of the implementation is straightforward, like an interpreter for a lambda calculus extended with booleans and integers [63]. It traverses the expression AST, recursively evaluating subexpressions as needed, while maintaining an environment map of names to values. The interpreter updates the environment when a variable is defined, and reads from it when a variable is referenced. Like most implementations, the Flix interpreter uses an environment to implement variables, instead of the more costly approach of substitutions, as defined in the lambda calculus semantics [2]. The interpreter also maintains a reference to the AST root so it can reference top-level definitions.

The Flix language supports more than just boolean and integer values. It also supports the unit value, characters, single- and double-precision floating point numbers, strings, tags, and tuples. Furthermore, Flix integers can have 8, 16, 32, 64, or an arbitrary number of bits. Therefore, the interpreter must also support these different values and types. Most of the types can be represented by classes in the Java standard library, but unit, tags, and tuples are represented by classes defined by the Flix run time.

Besides built-in unary and binary expressions, Flix also supports let-bindings, if-expressions, and pattern matching. Let-bindings and if-expressions are implemented directly, instead of desugaring let-bindings into lambda functions and applications. Pattern matching is not directly implemented; instead, pattern matching is compiled into lower-level primitives, as discussed in Section 3.1.1. The interpreter must then support the `CheckTag`, `GetTagValue`, and `GetTupleIndex` primitives.

A function call node in the AST contains a function reference, as well as a list of argument expressions. The interpreter simply evaluates the arguments from left to right, looks up the function reference from the top-level definitions, copies the environment map, extends the copied map with new bindings for the arguments, and then evaluates the function body with the new environment.

As discussed in Section 3.1.2, lambda functions are converted into closures, so the interpreter only needs to implement closure creation and closure calls. When the interpreter encounters a `MkClosure` node, it initializes a closure object with a reference to the implementing function and a list of the captured values. The `MkClosure` node contains the function reference as well as the names of the variables to capture. When a closure is called with arguments, the interpreter evaluates the arguments, and then looks up and calls the implementing function. The arguments to the implementing function are created by prepending the captured values to the closure argument values.

Finally, the interpreter must implement calls to external functions. An external function is represented as a `Hook` node in the AST, and contains an object implementing the external function. The function object can implement the `Invokable` or `InvokableUnsafe` interfaces, depending on whether the user prefers safety or performance. To implement the call, the interpreter evaluates the argument expressions and applies their values to the function object. If the function object implements the `Invokable` interface, then the interpreter must also wrap the arguments as `IValue`s and unwrap the return value.

3.3 Generating JVM Bytecode

The code generator back-end is the default implementation for the functional language because it has better performance compared to the interpreter back-end. The code generator consumes an `ExecutableAst` and produces JVM bytecode [51] that can be immediately loaded and executed. Flix uses the ASM library [14] to emit bytecode, which is the same library used by the Scala 2.12 back-end [22, 29].

ASM simplifies the bytecode generation process, since it takes care of many tedious details. For instance, ASM will automatically manage the constant pool, and can also compute each method's local variable array size and maximum operand stack height. All of this information must be embedded in the bytecode class file. ASM can also compute stack map frames, which are required metadata for the JVM bytecode verifier. Flix could manually compute this information, but this would add greater complexity to the back-end.

Emitting bytecode for most expressions is straightforward, since all operands and intermediate values are stored on the operand stack. For example, to generate code that adds two integer expressions `exp1` and `exp2`, the code generator first emits code for `exp1`, then for `exp2`, and then generates the `IADD` instruction—the `I` prefix indicates that the operands are 32-bit integers. At run time, the code for `exp1` is evaluated and its result is placed on the stack, and the same happens for `exp2`. Then `IADD` is evaluated, instructing the JVM to pop two 32-bit integers from the stack, add them, and then place the result back onto the stack.

The subsections below discuss code generation in further detail. The subsections include the organization of bytecode and how it is loaded and executed (3.3.1), how Flix values are implemented and represented at run time (3.3.2), integer semantics and how unary and binary expressions are compiled (3.3.3), and finally, the implementation of function, closure, and hook calls (3.3.4).

3.3.1 Organizing, Loading, and Executing Bytecode

Each Flix function is compiled as a static JVM method, and namespaces in Flix directly map to JVM packages and classes. For instance, `A.B.C/f()` is the fully qualified name for the function `f`, defined in the namespace `A.B.C`. The function is compiled to the static JVM method with fully qualified name `A.B.C.f()`, where `f()` is a static method defined within the class `C`, located within the package `A.B`. Flix constants are compiled as zero argument static methods; the implementation does not use fields. This avoids the need for initialization, and allows the run time to defer evaluation of error values. Finally, Flix variables within a function are implemented as local variables of a JVM method. Variables are accessed by the bytecode instructions `xSTORE` and `xLOAD`, where `x` is the prefix indicating the type of value to access. For example, `I` is the prefix for integer values.

The code generator produces bytecode one class at a time. Therefore, before code generation, all functions need to be grouped by their destination classes. To emit bytecode for a particular class, the code generator requires the fully qualified name of the target class, a list of Flix functions to compile, a map of function declarations, and a map of functional interfaces. The function declarations represent the functions and their types in the entire Flix program, and are needed since calling a function requires its full type signature. The functional interfaces represent the closures in the entire Flix program, and the implementation is discussed in Section 3.3.4.

Once a class has been emitted, the Flix implementation uses a Java `ClassLoader` to dynamically load it. A new instance of the class loader is created for each compilation of a Flix program, allowing multiple run-time instances of the same Flix program to exist. The class loader returns a reflection `Class` object, and each of the methods can be accessed as reflection `Method` objects. Flix extracts these `Method` objects and stores them on the AST, so that each function is implemented by both an expression subtree and a `Method` representing generated bytecode. The class loader also cooperates with garbage collection: if it is garbage collected, the classes are unloaded. A Flix AST references `Method` objects, which indirectly reference the class loader. Since there are no other references to the class loader, once the AST is garbage collected, the class loader will also be collected, which causes the classes to be unloaded.

When a Flix function is called, if there is no `Method` object, then the AST is evaluated by the interpreter. Otherwise, the `Method` is called with its `invoke` method. Since the `invoke` method must take boxed generic arguments, it will automatically unbox arguments and box return values when calling the underlying bytecode method. This allows Flix to generate methods that take and return primitive types.

Table 3.1: Flix types and their corresponding JVM types.

Flix type	**JVM type**	
	Primitive	Reference
`Unit`		`Value.Unit`
`Bool`	`boolean`	`java.lang.Boolean`
`Char`	`char`	`java.lang.Chaacter`
`Float32`	`float`	`java.lang.Float`
`Float64`	`double`	`java.lang.Double`
`Int8`	`byte`	`java.lang.Byte`
`Int16`	`short`	`java.lang.Short`
`Int32`	`int`	`java.lang.Integer`
`Int64`	`long`	`java.lang.Long`
`BigInt`		`java.math.BigInteger`
`Str`		`java.lang.String`
`Native`		`java.lang.Object`
`Tag`		`Value.Tag`
`Tuple`		`Value.Tuple`
`Lambda`		functional interface

3.3.2 Representing Flix Values

To avoid unnecessary allocations, Flix uses primitive values whenever possible. For example, a Flix `Bool` is represented as a JVM `boolean`, except for some cases when it must be boxed as a `java.lang.Boolean`. Some Flix values cannot be represented with primitive types, and so use reference types defined by the Java standard library or Flix. For instance, a Flix `Str` is represented as a `java.lang.String`, while a Flix tag is represented as a `Value.Tag`. Table 3.1 describes the mapping between Flix types and JVM types.

Literals are compiled by generating code that directly loads a constant onto the operand stack, or by generating code that explicitly constructs an object. For example, the literal `true` is compiled as an `ICONST_1` instruction, which instructs the JVM to push the value `1` onto the operand stack. The JVM represents `true` as the integer `1` and `false` as the integer `0`. Similarly, a string value is compiled into the constant pool and then loaded at run time. On the other hand, a `BigInt` value is compiled as code that explicitly constructs a `java.math.BigInteger` object.

For integer values, the process is more complicated. To reduce code size, the JVM provides instructions for loading different sizes of integers. For example, `ICONST_1` is an

instruction that loads the constant `1` at run time, and requires only a single byte. There are similar instructions for the values `-1` to `5`, inclusive. For larger values, the `BIPUSH` instruction takes an 8-bit integer (`byte`) argument and loads it onto the operand stack at run time. This requires two bytes of instruction code. Similarly, `SIPUSH` takes a 16-bit integer (`short`) argument and loads it at run time, requiring three bytes of instruction code. Integers that require more than 16 bits must be compiled into the constant pool.

Tag and tuple values are represented by classes defined by Flix. Furthermore, they are compound values: a tag contains a single inner value, while a tuple contains an array of values. A tag is represented by the `Value.Tag` class, and can be constructed by calling the helper method `Value.mkTag` with the tag name and the inner value. Similarly, a tuple is represented by the `Value.Tuple` class. However, constructing a `Value.Tuple` involves allocating an array and initializing each of its elements.

Since tags and tuples are generic in the values they can contain, Flix uses type erasure, representing inner values as `java.lang.Object` values. Thus, primitive values need to be boxed when constructing tags and tuples, and unboxed when they are extracted. Reference values do not need to be boxed, but they need to be cast to their proper types when extracted. All of this is done automatically at run time, since the code generator emits the necessary boxing and unboxing code.

3.3.3 Implementing Unary and Binary Expressions

Unary and binary expressions in Flix can be classified as arithmetic, bitwise, comparison, or logical expressions. Each of the types of expressions have their own implementation details. However, arithmetic and bitwise expressions require extra consideration because they manipulate integer values, and the Flix and JVM semantics are slightly different.

Both Flix and the JVM support 8-bit, 16-bit, 32-bit, and 64-bit signed integers, representing them in signed two's complement notation. An integer with N bits can represent values from -2^{N-1} to $2^{N-1} - 1$, inclusive, where a positive number x is represented as itself, and a negative number y is represented as $2^N - y$. However, on the JVM, 8-bit and 16-bit integers are sign extended to 32 bits, effectively changing their types to 32-bit integers. Thus, arithmetic expressions on 8-bit and 16-bit integers are actually arithmetic expressions on 32-bit integers, and return 32-bit integers. The following Java method will not compile because it returns an `int`, which does not match the declared return type of `byte`:

```
public static byte f(byte x, byte y) {
    return x + y; // compilation error: 'x + y' has type int, not byte
}
```

Flix is different in this respect. An arithmetic expression in Flix returns an integer with the same number of bits as its operands, because 8-bit and 16-bit integers behave as if there is no sign extension. (In practice, they are sign extended because they have to be represented on the JVM.) This is an intentional design decision of Flix to ensure consistency, as integer values will always remain the type they were originally declared as.

The different semantics will produce different values if overflow occurs; in other words, if there is an insufficient number of bits to represent the result of an operation. For both the JVM and Flix, if an operation on 32-bit (or 64-bit) integers overflows, the result is taken to be the 32 (or 64) low-order bits of the mathematical result, performed in two's complement notation with a sufficient number of bits. Operations involving 8-bit and 16-bit integers will never overflow on the JVM, because they are sign extended to 32 bits. However, in Flix, operations behave as if there is no sign extension, so overflow is possible. A user who is concerned with overflow should consider using an integer type with more bits, or even the `BigInt` type which has arbitrary precision.

As an example of overflow, consider adding two 8-bit integers with value `64`. With JVM semantics, the result is the 32-bit value `128`. However, with Flix semantics, the result is the 8-bit value `-128` because of overflow. (Since the operation takes place on the JVM, the result will be sign extended to 32 bits.) The implementation task, then, is to generate code that follows Flix semantics while using operations that follow JVM semantics.

The subsections below discuss the implementation of arithmetic, bitwise, comparison, and logical expressions. While the comparison and logical expressions do not need to account for the different integer semantics between Flix and the JVM, they have other issues to consider. The discussion will use the notation

$$\texttt{00000000\_00000000\_00000000\_00000000}_2$$

to represent binary, with bits grouped by eight, and the notation

$$\texttt{99}_{10}$$

to represent decimal.

Arithmetic Expressions

Flix supports the following arithmetic operations: unary minus and plus, addition, subtraction, multiplication, division, modulo, and exponentiation. Exponentiation is implemented as a call to the Java standard library. The library function `java.lang.Math.pow` requires operands to be cast to `double` and the result to be cast from `double`.

The other arithmetic operations are implemented in bytecode; however, the result must be consistent with Flix semantics. Specifically, values should behave as if they were not sign extended, and operations should return values of the same type as the operands. Consider the unary minus operation and the 8-bit operand

$$10000000_2 = -128_{10}$$

The mathematical result is 128_{10}, which cannot be represented in 8 bits. Following Flix semantics, the result is the 8 low-order bits of the mathematical result represented in two's complement notation, which happens to be

$$10000000_2 = -128_{10}$$

However, on the JVM, the operand is sign extended to

$$11111111\_11111111\_11111111\_10000000_2 = -128_{10}$$

Applying unary minus, the result is

$$00000000\_00000000\_00000000\_10000000_2 = 128_{10}$$

which is the correct mathematical result, but the incorrect result for an 8-bit integer. To obtain the correct representation, the code generator emits the `I2B` instruction, which will truncate the value to 8 bits and then sign extend it. This gives the correct representation:

$$11111111\_11111111\_11111111\_10000000_2 = -128_{10}$$

For 16-bit integers, the result must be truncated to 16 bits and then sign extended. Note that the interpreter must also handle the different integer semantics, which is done by casting results to `byte` (to truncate to 8 bits) or to `short` (to truncate to 16 bits).

The process is the same for binary arithmetic operations: results need to be truncated and then sign extended. Adding two 8-bit integers should return an 8-bit integer, even though it is sign extended to 32 bits. Consider adding two 8-bit integers with value `64`:

$$\begin{array}{rl} & 01000000_2 = 64_{10} \\ + & 01000000_2 = 64_{10} \\ \hline & 10000000_2 = -128_{10} \end{array}$$

The mathematical result is 128_{10}, but with only 8 bits, the representation requires the result to be -128_{10}. However, on the JVM, the 8-bit integers are sign extended to 32-bit integers, which gives the following result:

```
   00000000_00000000_00000000_01000000₂ =   64₁₀
 + 00000000_00000000_00000000_01000000₂ =   64₁₀
-----------------------------------------------
   00000000_00000000_00000000_10000000₂ =  128₁₀
```

Again, the code generator must emit code that truncates and sign extends the result, to give the correct representation:

$$\texttt{11111111\_11111111\_11111111\_10000000}_2 = \texttt{-128}_{10}$$

Bitwise Expressions

Flix supports the following bitwise operations: complement, and, or, exclusive-or, left shift, and right shift. Most of these expressions do not require any truncation, because bitwise expressions operate on bits rather than the value represented by those bits.

A complement operation is implemented as an exclusive-or, since there is no bytecode instruction that inverts the bits of a value. However, the complement of an N-bit value is the same as the exclusive-or of that value and a second value of N 1s. For instance, the complement of

$$\texttt{11000011}_2$$

is

$$\texttt{00111100}_2$$

and can be implemented as an exclusive-or:

```
   11000011₂
 ^ 11111111₂
--------------
   00111100₂
```

On the JVM, the operands are sign extended to 32 bits, but the 8 low-order bits are the same. Truncating and sign extending the result makes no difference. Consider the previous example, but with the operands sign extended:

```
   11111111_11111111_11111111_11000011₂
 ^ 11111111_11111111_11111111_11111111₂
-----------------------------------------
   00000000_00000000_00000000_00111100₂
```

The bitwise-and and bitwise-or operations also do not require any additional truncations or sign extensions. Consider, for example

```
   11110000₂
 & 11000000₂
--------------
   11000000₂
```

As with exclusive-or, the operands are sign extended on the JVM, but the 8 low-order bits remain the same. Truncating and sign extending the result also has no effect, as demonstrated below:

```
   11111111_11111111_11111111_11110000₂
 & 11111111_11111111_11111111_11000000₂
---------------------------------------
   11111111_11111111_11111111_11000000₂
```

However, the bitwise left shift operation requires truncation and sign extension, because the operation affects the high-order bits. Consider the following example, where `x` and `y` represent unknown values:

```
   x000y000₂
<< 00000100₂
--------------
   y0000000₂
```

Due to sign extension, the result is represented as

```
   yyyyyyyy_yyyyyyyy_yyyyyyyy_y0000000₂
```

However, on the JVM, all operands are sign extended to 32 bits, giving the actual result:

```
   xxxxxxxx_xxxxxxxx_xxxxxxxx_x000y000₂
<< 00000000_00000000_00000000_00000100₂
---------------------------------------
   xxxxxxxx_xxxxxxxx_xxxxx000_y0000000₂
```

The code generator must emit code to truncate and sign extend the result, to achieve the correct representation. Note that the value for `x` was irrelevant; however, the value for `y` affects the sign extension.

Finally, the bitwise right shift does not require any truncation for the result. Consider, for instance

```
   x0000000₂
>> 00000100₂
--------------
   xxxxx000₂
```

On the JVM, the operands are sign extended, giving:

```
   xxxxxxxx_xxxxxxxx_xxxxxxxx_x0000000₂
>> 00000000_00000000_00000000_00000100₂
---------------------------------------
   xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxx000₂
```

In this example, the values of the high-order bits are completely determined by the value of `x`. Therefore, it is not necessary to truncate or sign extend the result.

Comparison Expressions

In Flix, integers, floating point numbers, and characters support all six comparison operators: equal (`EQ`), not equal (`NE`), greater than (`GT`), greater than or equal (`GE`), less than (`LT`), and less than or equal (`LE`). However, unit, booleans, strings, tags, and tuples only support equality (`EQ` and `NE`). Primitive values are compared by using the JVM comparison instructions, while reference values are compared by emitting a call to a method, such as `Object.equals`.

Comparisons of primitive values are implemented as a branch on the *negated* condition: if the negated condition succeeds (in other words, if the original condition fails), control jumps to the false branch and returns `0`. Otherwise, control continues to the true branch and returns `1`. The JVM provides different comparison instructions depending on the involved types. Booleans, 8-bit integers, 16-bit integers, and characters are represented as 32-bit integers, so they use the integer comparison instructions. These instructions have the form `IF_ICMPyy` where `yy` is one of the six comparison operators. `IF_ICMPyy` takes a label as an argument and jumps to it if the comparison succeeds.

Comparing 64-bit integers requires a pair of instructions, `LCMP` and `IFyy`, which are used together to achieve the same effect as `IF_ICMPyy` for 32-bit integers. `IFyy` compares the result of `LCMP` with `0` and jumps to the given label if the comparison succeeds. In pseudocode, `LCMP` compares the values `v1` and `v2` with the following semantics:

```
if      (v1 >  v2)  1
else if (v1 == v2)  0
else if (v1 <  v2) -1
```

Finally, to compare floating point numbers, the JVM provides `FCMPG` and `FCMPL` for single-precision floating point numbers, and `DCMPG` and `DCMPL` for double-precision floating point numbers. These instructions are similar to `LCMP`, except that the `G` and `L` suffix affects how `NaN` is handled—all comparisons involving `NaN` must fail.

Logical Expressions

Flix supports the following logical operators: not, and, or, implication, and biconditional. Logical expressions are the simplest of the unary and binary expressions to implement. However, like Java, logical expressions use short-circuit evaluation. The second expression of a logical-and is only evaluated if the first expression evaluates to `true`, and the second expression of a logical-or is only evaluated if the first expression evaluates to `false`.

The bytecode instructions do not have direct support for short-circuit evaluation, so the expression `EXP1 && EXP2` is compiled as

```
if (EXP1) EXP2 else 0
```

and the expression `EXP1 || EXP2` is compiled as

```
if (EXP1) 1 else EXP2
```

The implication (`==>`) and biconditional (`<==>`) operators are syntactic sugar introduced to Flix. The code generator simply rewrites them before emitting code.

```
EXP1 ==> EXP2
```

is rewritten to

```
!EXP1 || EXP2
```

and

```
EXP1 <==> EXP2
```

is rewritten to

```
EXP1 == EXP2
```

3.3.4 Calling Functions, Closures, and Hooks

The target `f` of a call can be a Flix function, closure, or a hook. The simplest case is when `f` is a function reference. The code generator looks up `f` in the declarations map to get its full type signature, emits code to evaluate the arguments of the call, and then emits a static method invocation instruction. The method invocation requires both the name and full type signature of the target, and assumes argument values are on the operand stack. The implementation is more complicated for closure and hook calls, and is discussed below.

Calling Closures

One approach for implementing closures is to use anonymous classes. Each closure is represented by a class, with fields for captured variables and a method that implements the closure function. At run time, a closure object is created by calling the constructor with the captured variables, and a closure is called by invoking its method. However, this approach increases code size, as every lambda expression requires an anonymous class. An alternate approach, taken by Java 8 and Scala 2.12 [22, 31], is to use `invokedynamic` to create closure objects [67]. A closure object implements a functional interface, which declares a single abstract method representing the implementation of the closure function. A closure is called by invoking its interface method.

An `invokedynamic` instruction represents a *dynamic call site*, which allows the target method to be determined at run time. Initially, a dynamic call site is unlinked, meaning the target method is unknown. The JVM will perform linking by calling the *bootstrap method* specified by the `invokedynamic` instruction. Linking happens just before the first invocation of a dynamic call site, and subsequent invocations will skip the bootstrap method call and directly call the linked method.

When the code generator encounters a `MkClosure` node, it emits an `invokedynamic` call, with `java.lang.invoke.LambdaMetafactory.metafactory` as the bootstrap method. `metafactory` takes both static and dynamic arguments. The static arguments represent the functional interface implemented by the closure and a method handle for the implementing method of the closure, while the dynamic arguments represent the captured values. When the closure is created for the first time, the `metafactory` generates bytecode for an anonymous class that implements the functional interface, links its constructor to the dynamic call site, and then calls the constructor with the values of the captured variables. Subsequent creations of that particular closure will bypass linking and directly call the constructor. To call a closure, the code generator emits code for an interface invocation. The closure will then call its implementing method with the captured variables and closure arguments.

Each closure object implements a functional interface and is invoked through that interface. However, the interface must be provided by the implementation. The Java standard library provides a small collection of specialized interfaces, while at the opposite extreme, the Scala standard library provides a much larger collection of generic interfaces. The Java interfaces are too limited, while the Scala interfaces are too general, so Flix takes its own approach. Specifically, the code generator creates only the functional interfaces that are required. This avoids generics (and boxing and unboxing), since the interfaces are specialized to the closure types that appear in the program.

The functional interfaces are generated shortly before methods are compiled. A visitor traverses the entire AST, collects the type signatures of every closure used in the program, and then generates a name for each unique type. This creates the functional interfaces map, which will later be used in code generation. Next, the code generator compiles an interface for each name. Each interface is annotated with `@FunctionalInterface` and declares a single abstract method `apply`, with the type signature of the closure it represents. Once all functional interfaces have been generated, the code generator can compile the methods. To emit code for a closure call, the code generator will look up the appropriate functional interface from the map and then emit the interface call to `apply`.

Calling Hooks

The Flix run-time object maintains a map of all hooks and provides the instance methods `invoke` and `invokeUnsafe` for calling external functions. These methods are part of the public API, but their main purpose is to implement hook calls in the generated bytecode. Both `invoke` and `invokeUnsafe` take a string representing the name of the hook to call, as well as an array representing the argument values. Argument values to `invoke` need to be wrapped as `IValue`s, and the return value needs to be unwrapped.

Because `invoke` and `invokeUnsafe` are instance methods, the Flix run-time object must be available to the generated code. Therefore, each bytecode class is generated with a static field that refers to the Flix object. The field is initialized to `null`, but after the class is dynamically loaded, the Flix implementation sets the field to itself.

Chapter 4

Evaluation

This chapter discusses the testing and evaluation of the Flix back-ends. The first group of tests ensures correctness of the interpreter and code generator (4.1), while the second group evaluates the performance of the back-ends (4.2).

4.1 Testing for Correctness

The first group of tests, mostly implemented in the ScalaTest framework [79], ensures correctness of all aspects of the implementation. Each test is executed with the interpreter and code generator, and results are compared against an expected result. Currently, there are over 500 tests for the functional language implementation.

Each test is written as a small but complete Flix program, thus testing the entire front-end—the parsing, weeding, name resolution, and type checking phases—as well as the back-end. Originally, tests would manually construct ASTs, but this was very tedious and brittle, as the `ExecutableAst` was still evolving. It was also possible to construct invalid ASTs, leading to unexpected results. The current strategy of writing complete Flix programs provides the benefit of testing the front-end, is easier to write and maintain, and guarantees that all input to the back-end is valid.

A larger and more realistic test is implemented outside the ScalaTest framework. This test is a Flix implementation of the Strong Update analysis, a points-to analysis for C programs [50]. The Strong Update analysis achieves better precision by propagating singleton sets flow-sensitively, while maintaining efficiency by propagating larger sets flow-insensitively. The test runner is a custom Ruby script that compares the Flix implementation

of the Strong Update analysis (both interpreted and compiled) to a Datalog implementation, which is run on the DLV solver [49]. The analysis requires a constant propagation lattice, which can be directly expressed in Flix. However, because Datalog does not support user-defined lattices, the Datalog implementation must embed the lattice in a power set, which is expensive. The script uses the SPEC CPU2000 and CPU2006 integer benchmarks [75] as analysis inputs, and ensures both implementations return result sets of the same size. The script only checks the size of the result sets and not their contents, since it is very unlikely that the two implementations return sets of the same size but with different contents.

4.2 Benchmarking for Performance

The remaining tests evaluate the performance of the interpreter and the generated code, and can be grouped into three categories. The first category consists of purely functional toy benchmarks. These programs do not require the Flix solver and can be compared with implementations in other languages. The second category includes toy benchmarks that use both functional and logic code, and therefore require the Flix solver. As the solver would require significant effort to re-implement in other languages, these benchmarks only compare the interpreter and code generator back-ends, and not other implementations. Finally, the last category consists of a real-world static analysis that contains both functional and logic code. All evaluation code is available on GitHub[1].

The remainder of this section discusses the different benchmarks (4.2.1), the testing environment and implementation languages (4.2.2), and the results (4.2.3).

4.2.1 Description of Benchmarks

There are six performance benchmarks. `fib`, `nbody`, and `pidigits` are purely functional benchmarks; `matrixmult` and `shortestpaths` contain both functional and logic code; and `strongupdate` is a real-world static analysis. The benchmarks are described below.

`fib` This is a naïve implementation of the Fibonacci function, with exponential running time. For a given input n, it computes the n^{th} Fibonacci number.

`nbody` Adapted from the Computer Language Benchmarks Game [1], this program is an N-body simulation. For the given input m, it models the dynamics of the Sun and the four gas giants for m iterations.

[1] https://github.com/mhyee/flix-experiments

`pidigits` Another program adapted from the Computer Language Benchmarks Game, this benchmark computes the first n digits of π, for a given n, using an unbounded "spigot" algorithm [30]. The implementation requires arbitrary-precision arithmetic.

`matrixmult` This program multiplies two randomly generated $n \times n$ matrices, using the naïve algorithm with cubic running time. The implementation requires the Flix solver, as it models matrices with relations and lattices. Although the model is imperfect, it is sufficient to implement matrix multiplication. This benchmark was only implemented in Flix, as porting the solver was not feasible for the evaluation.

`shortestpaths` This benchmark is based on the Floyd-Warshall algorithm for all-pairs shortest paths [27]. However, it computes the *length* of the shortest path, rather than the path itself. The input to the benchmark is a random graph of n vertices. The edges of the graph are modelled as a relation, and the distances are modelled as a lattice over the natural numbers. Like `matrixmult`, this benchmark was only implemented in Flix.

`strongupdate` The Strong Update analysis is a points-to analysis for C programs, as described previously. The Flix implementation is compared to a Datalog implementation, as well as a handwritten C++ implementation.

4.2.2 Testing Environment and Languages

With one exception, all benchmarks were performed on a desktop workstation with 16 GB of RAM and an AMD FX-8320 processor clocked at 3.50 GHz, running Ubuntu 12.04.5 LTS. The exception is the C++ implementation of the Strong Update analysis, whose results were taken from the original Strong Update paper. The other tests were managed by a custom Ruby script, which ran each benchmark individually and recorded the average execution time over 20 trials. The script also monitored the programs, terminating any run that exceeded the 15 minute timeout, and properly reported any errors that would occur.

The benchmarks that contain only functional code (`fib`, `nbody`, and `pidigits`) were implemented in Flix, Ruby, Scala, Java, and C++. Each program was written in an idiomatic style, with no attempt to optimize the programs or make them as similar to the Flix versions as possible. A brief discussion of each language and implementation follows.

Flix As a pure functional language, Flix has certain features that limit its performance. For example, without mutation, all data structures must be copied rather than updated in

place. Furthermore, iteration is implemented as recursion, but tail call optimization is not yet implemented. Both the interpreter and generated bytecode were run on the Java HotSpot™ 64-Bit Server VM. The JVM was configured to have a maximum heap size of 8 GB and a thread stack size of 128 MB. Arbitrary-precision integers are built into the language and implemented using the Java standard library.

Ruby The benchmarks were implemented in Ruby 2.3.1 and run on the default YARV bytecode interpreter [68]. The benchmarks use mutation and iteration with while loops. Arbitrary-precision integers are provided by the standard library, which uses the GNU Multiple Precision Arithmetic Library (GMP) [28].

Scala The Scala [61] benchmarks were written in Scala 2.11.8 and run on the Java HotSpot™ 64-Bit Server VM, which was configured with a maximum heap size of 8 GB and a thread stack size of 32 MB. The programs were written in a pure functional style that resembles Flix, rather than an imperative style that resembles Java. In other words, all data structures are immutable and must be copied, and the benchmarks use recursion. However, tail call optimization was not disabled. Arbitrary-precision integers are provided by the Java standard library.

Java The Java [33] benchmarks were implemented in Java 1.8 and run on the Java HotSpot™ 64-Bit Server VM, which was configured with a maximum heap size of 8 GB and a thread stack size of 32 MB. The programs were implemented in an object-oriented style, allowing mutation and while loops. Arbitrary-precision integers are provided by the standard library.

C++ The benchmarks were implemented in C++14 [39] and compiled under GCC 6.1.0 with the highest optimization level (`-O3`). Like the Java programs, the C++ programs were implemented in an object-oriented style, allowing mutation and while loops. Arbitrary-precision integers are provided by GMP.

4.2.3 Discussion of Results

Overall, the generated bytecode for Flix runs faster than the interpreter. The speedup is considerable for the `fib` and `nbody` benchmarks, and minimal for the `pidigits`, `matrixmult`, and `shortestpaths` benchmarks. Furthermore, as demonstrated by the graphs, Flix is able to scale with the other languages. Although C++ is easily the fastest, compiled Flix is comparable to Scala and Java for `fib` and `pidigits`. For `strongupdate`, both the interpreted and compiled Flix implementations are faster than the Datalog implementation, though substantially slower than the handwritten C++ implementation.

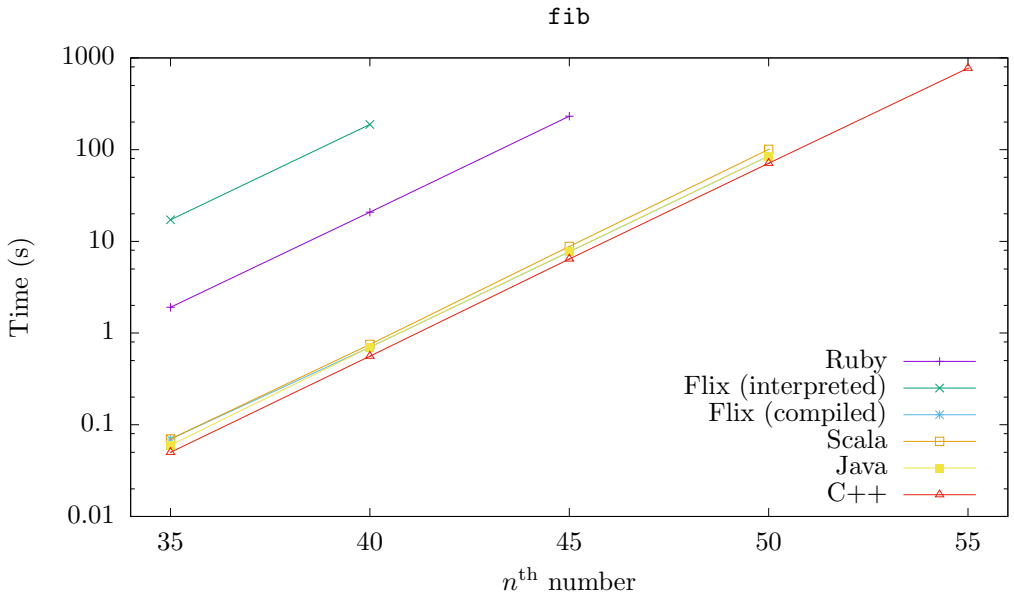

Figure 4.1: Time taken to compute the n^{th} Fibonacci number, in seconds.

The results of the `fib` benchmark are summarized in Figure 4.1 and Table 4.1. On average, the compiled Flix program is over 250x faster than the interpreted Flix program, even beating Scala. Since Fibonacci is a very simple function, the generated bytecode for Flix, Scala, and Java is similar. However, Scala is slower because the program was written as instance methods of a singleton object, rather than static methods. The Ruby implementation is slow, though much faster than interpreted Flix, which uses an AST interpreter instead of a bytecode interpreter. Finally, as expected, C++ is the fastest.

Figure 4.2 and Table 4.2 summarize the `nbody` benchmark. Here, both Flix implementations are the slowest, though the compiled version is, on average, 17x faster than the interpreted version. The `nbody` benchmark is currently the largest and most complicated purely functional program written in Flix, and highlights many inefficiencies. For instance, the lack of tail call optimization increases the stack space requirements, until the program runs out of memory. In addition, structures in the simulation need to be represented as tags and tuples, which box their values and increases memory consumption. Furthermore, maintaining and copying the environment maps in the Flix interpreter adds significant overhead. These inefficiencies allow Ruby to outperform Flix. Scala and Java are still faster and have similar performance, though Java is slightly faster. Finally, C++ is the fastest, as the compiler emits vector instructions to optimize the computations.

Table 4.1: Summary of performance results for computing the n^{th} Fibonacci number. Timeout means more than 15 minutes.

	Ruby	**Flix**		**Scala**	**Java**	**C++**
n	Time (s)	Interpreted (s)	Compiled (s)	Time (s)	Time (s)	Time (s)
35	1.91	17.15	0.07	0.07	0.06	0.05
40	20.81	187.95	0.70	0.75	0.70	0.56
45	231.58	timeout	7.70	8.79	7.75	6.42
50	timeout	-	84.98	100.58	85.05	71.08
55	-	-	timeout	timeout	timeout	772.95
60	-	-	-	-	-	timeout

Table 4.2: Summary of performance results for computing m thousand iterations of the N-body simulation. OOM means out of memory (stack overflow).

m	**Ruby**	**Flix**		**Scala**	**Java**	**C++**
(thousands)	Time (s)	Interpreted (s)	Compiled (s)	Time (s)	Time (s)	Time (s)
16	0.20	14.10	0.63	0.06	0.03	0.01
32	0.40	27.76	1.05	0.11	0.06	0.03
64	0.80	57.95	2.95	0.19	0.12	0.05
128	1.59	116.24	8.86	0.34	0.23	0.10
256	3.20	246.21	29.27	0.62	0.45	0.21
512	6.36	OOM	78.77	1.17	0.89	0.41
1,024	12.76	-	OOM	2.31	1.79	0.84

Table 4.3: Summary of performance results for computing the first n thousand digits of π. Timeout means more than 15 minutes. OOM means out of memory (stack overflow).

n	**Ruby**	**Flix**		**Scala**	**Java**	**C++**
(thousands)	Time (s)	Interpreted (s)	Compiled (s)	Time (s)	Time (s)	Time (s)
4	1.71	1.78	1.53	1.37	1.40	0.21
8	7.17	6.15	5.66	4.97	5.14	0.87
16	30.41	22.50	21.43	20.45	21.11	3.75
32	126.44	93.58	87.24	86.61	83.15	16.14
64	550.68	OOM	371.97	367.24	350.07	79.59
128	timeout	-	timeout	timeout	timeout	363.01
256	-	-	-	-	-	timeout

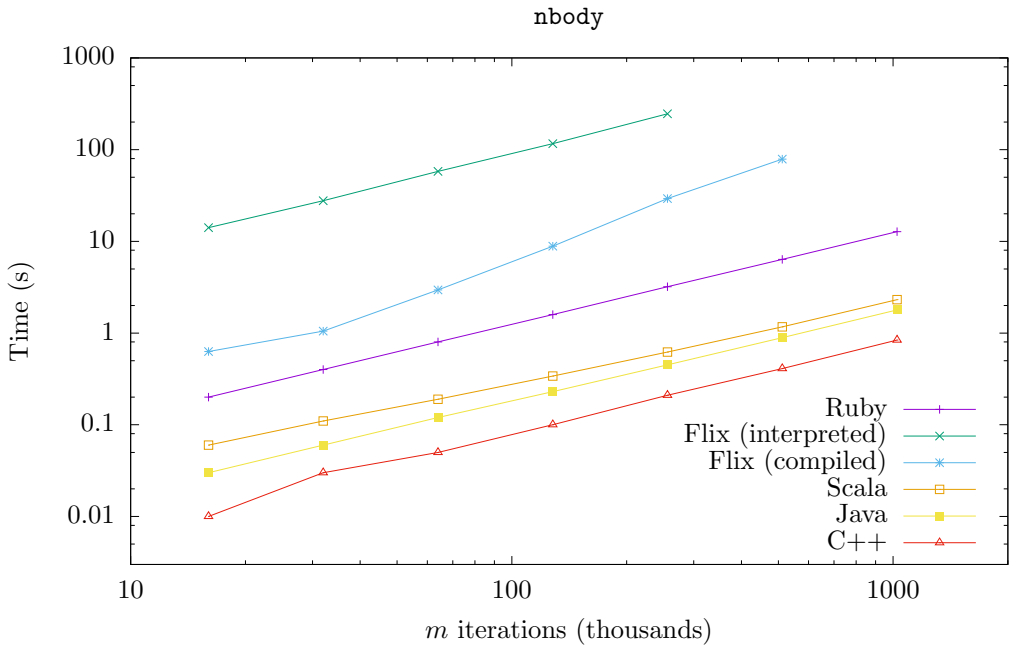

Figure 4.2: Time taken to compute m thousand iterations of the N-body simulation, in seconds.

For the `pidigits` benchmark, C++ is clearly the fastest, but the remaining implementations have similar performance results. Figure 4.3 and Table 4.3 show that Java and Scala are very close, though both are slightly faster than compiled Flix. Most of the computation in `pidigits` involves arbitrary precision arithmetic, and Java, Scala, and Flix use the same library (`java.math.BigInteger`), which explains why the results are similar. The interpreted Flix program runs out of stack memory, as it does not have tail call optimization. On the other hand, the compiled Flix program is more memory efficient, and times out before it can run out of memory. Interestingly, the Ruby bytecode interpreter was slower than the Flix AST interpreter.

The next two benchmarks, `matrixmult` and `shortestpaths` were only implemented in Flix, since they require the solver. The `matrixmult` results are summarized in Figure 4.4 and Table 4.4, while the `shortestpaths` results are summarized in Figure 4.5 and Table 4.5. Both Flix back-ends performed very similarly, with the generated code being slightly faster than the interpreted code. The difference is minimal because the functional code has very little impact on the overall performance—the logic code and solver consume most of the execution time.

Table 4.4: Summary of performance results for multiplying two randomly generated $n \times n$ matrices. Timeout means more than 15 minutes.

n	**Interpreted (s)**	**Compiled (s)**
8	0.12	0.09
16	0.25	0.18
32	0.57	0.50
64	2.49	2.02
128	17.00	14.02
256	130.05	106.16
512	timeout	timeout

Table 4.5: Summary of performance results for computing the shortest paths in a random graph of n vertices. Timeout means more than 15 minutes.

n	**Interpreted (s)**	**Compiled (s)**
8	0.20	0.15
16	0.18	0.15
32	0.77	0.65
64	0.99	0.84
128	0.78	0.63
256	15.32	14.29
512	8.97	8.86
1024	30.37	29.89
2048	430.80	426.30
4096	timeout	timeout

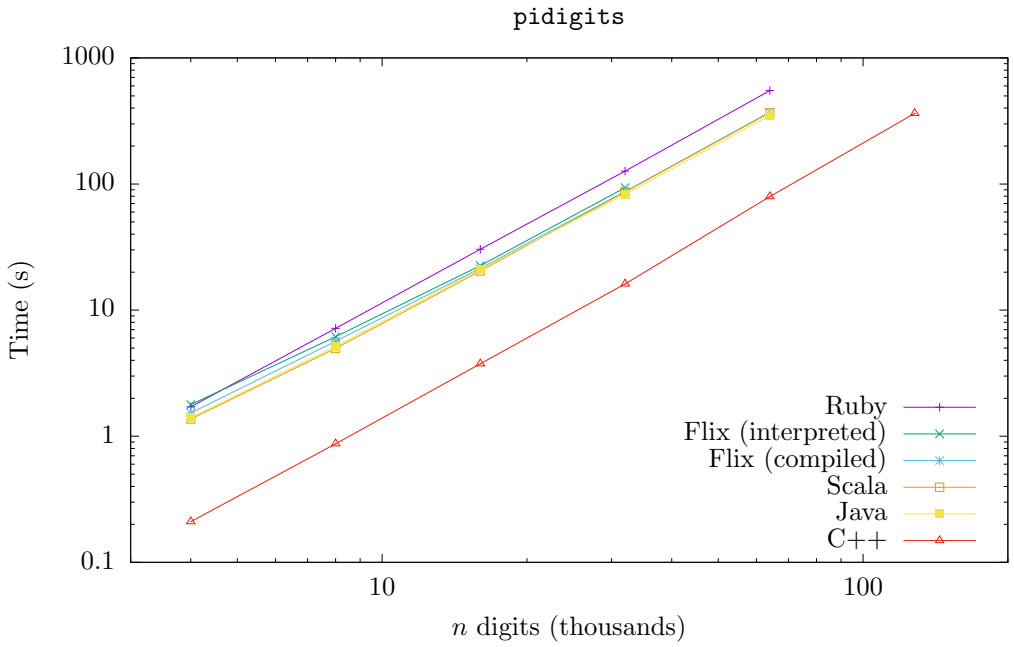

Figure 4.3: Time taken to compute the first n thousand digits of π, in seconds.

Finally, `strongupdate` represents a real-world points-to analysis that one might write in Flix. The benchmark compares four different implementations of the analysis: one in Datalog and executed on the DLV solver, an interpreted Flix implementation, a compiled Flix implementation, and a handwritten C++ implementation. The results for the C++ analyzer were taken from the original Strong Update paper. The SPEC CPU2000 and CPU2006 integer benchmarks were used as analysis inputs.

The results are summarized in Figure 4.6 and Table 4.6. Both implementations of Flix are faster than the Datalog implementation, showing that the analysis is more efficient with user-defined lattices. On average, the interpreted Flix analysis is 3.7x faster than Datalog, and the generated Flix bytecode is 1.7x faster than the interpreter. However, the C++ implementation is 126x faster than compiled Flix. Note that this difference in speed is much larger than what could be accounted for by the different testing environments.

The main reason the C++ implementation is faster is that it can significantly reduce memory usage. Within the analysis, some elements of the lattice occur much more frequently than others, and the C++ implementation uses a custom data structure to implicitly represent these common elements. However, in Flix, the elements are explicitly stored, which substantially increases memory consumption.

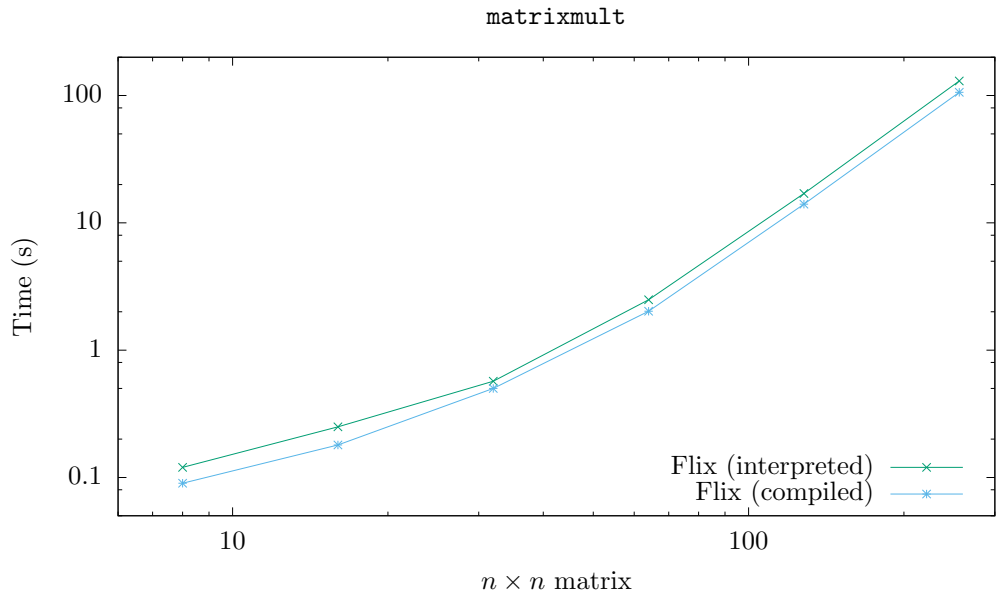

Figure 4.4: Time taken to multiply two randomly generated $n \times n$ matrices, in seconds.

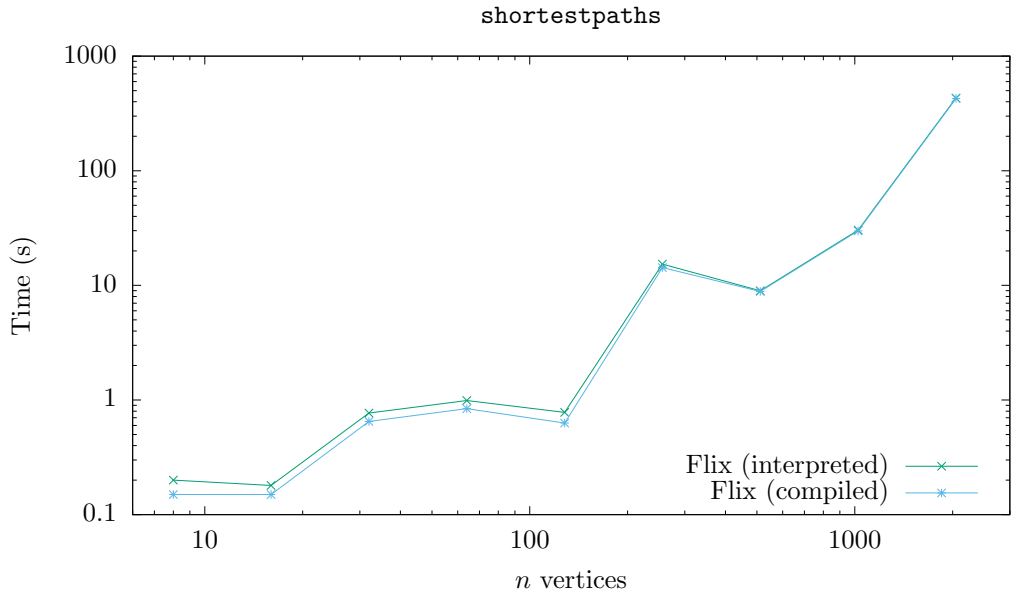

Figure 4.5: Time taken to compute the shortest paths in a graph of n vertices, in seconds.

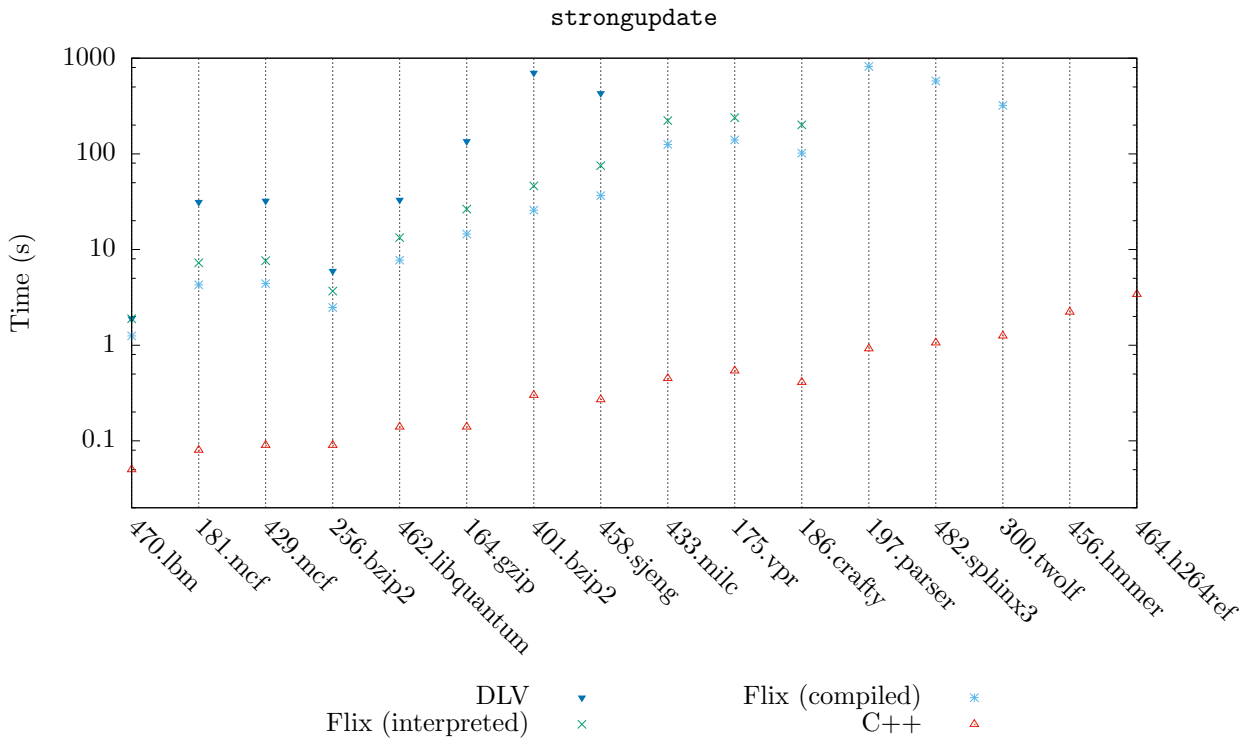

Figure 4.6: Time taken to run the Strong Update analysis, in seconds.

Table 4.6: Summary of performance results for the Strong Update analysis. Timeout means more than 15 minutes.

Benchmark			**DLV**	**Flix**		**C++**
Program	kSLOC	Input Facts	Time (s)	Interpreted (s)	Compiled (s)	Time (s)
470.lbm	1.2	1,205	1.90	1.89	1.25	0.05
181.mcf	2.5	3,377	31.32	7.26	4.27	0.08
429.mcf	2.7	3,392	32.26	7.64	4.40	0.09
256.bzip2	4.7	5,017	5.97	3.68	2.48	0.09
462.libquantum	4.4	6,196	33.02	13.33	7.75	0.14
164.gzip	8.6	9,259	135.66	26.45	14.52	0.14
401.bzip2	8.3	11,844	702.95	46.28	25.79	0.30
458.sjeng	13.9	20,154	430.49	75.36	36.62	0.27
433.milc	15.0	22,147	timeout	222.43	125.24	0.45
175.vpr	17.8	25,977	-	238.69	139.99	0.54
186.crafty	21.2	32,189	-	201.28	101.82	0.41
197.parser	11.4	32,606	-	timeout	818.44	0.92
482.sphinx3	25.1	42,736	-	-	576.86	1.06
300.twolf	20.5	44,041	-	-	320.86	1.25
456.hmmer	36.0	68,384	-	-	timeout	2.22
464.h264ref	51.6	89,898	-	-	-	3.41
seven more benchmarks						

Chapter 5

Related Work

This chapter explores work related to Flix and functional language implementations. It discusses traditional static analysis frameworks (5.1), Datalog, static analysis tools that use Datalog, and languages related to Flix (5.2), and finally, implementation techniques (5.3).

5.1 Static Analysis Frameworks

There has been much prior work in designing static analysis frameworks. The program analyzer generator PAG takes a high-level specification of data types, lattices, and transfer functions, and generates C code for an analyzer [56]. Soot is a framework for analyzing Java bytecode and has been widely used for other tools and front-ends [78]. The T. J. Watson Libraries for Analysis (WALA), implemented in Java, are tools for static program analysis [25]. They support Java and JavaScript, and features include class hierarchy analysis, points-to analysis, and call graph construction. Hoopl is a library for implementing dataflow analyses and transformations, written in Haskell [64]. Frama-C is a software analysis platform for C programs, written in OCaml [19]. The OPAL framework, implemented in Scala, is a software product line for creating Java bytecode analyzers [23].

Many of these frameworks include implementations of specific analyses, which could be written in Flix rather than a general-purpose language. This would make it easier to understand and ensure correctness of the analysis, and to separate the specification of the analysis from the implementation of the fixed-point algorithm. Furthermore, many of these frameworks make certain assumptions about the analysis being implemented, such as requiring a control-flow graph, but Flix is general and solves least fixed point problems.

5.2 Datalog

Datalog is a declarative programming language with roots in logic programming and deductive databases. It is a syntactic subset of Prolog and can be used as a query language for relational databases; however, unlike core SQL, Datalog queries support recursion. Ceri et al. [16] provide a thorough introduction to Datalog, which discusses its syntax and semantics and shows how Datalog relates to the relational algebra. Huang et al. [36] provide a more succinct tutorial, and also suggest that there is renewed interest in using Datalog for a variety of applications, including data integration, networking, and program analysis.

Many static analysis tools involve Datalog. bddbddb is a Datalog implementation that represents relations as binary decision diagrams, and has been used for context-sensitive pointer analysis [47, 80]. CodeQuest is a tool for querying source code that compiles Datalog to SQL [34], and QL is an object-oriented query language that compiles to Datalog [12]. Doop is a Datalog framework for scalable and precise context-sensitive points-to analyses [13, 71, 72]. Alpuente et al. [4] describe techniques for efficiently evaluating Datalog queries, tailored to object-oriented program analysis. Allen et al. [3] introduce a points-to framework in Datalog that can scale to the OpenJDK™ library. Finally, Soufflé is a framework that synthesizes C++ analyzers from Datalog specifications [43, 69].

Flix shares some similarities with Bloom^L, a language for ensuring consistency in distributed programs [17]. Bloom^L extends Bloom, a language based on Datalog, with lattices and monotone functions. While Flix programs compute the minimal model of a fixed-point problem and terminate, Bloom^L programs run continuously.

Another similar language is Datafun, a functional language that allows Datalog-style programming and tracks monotonicity with types [9]. Like Flix, Datafun is more expressive than Datalog, allowing programmers to write monotone functions. However, while Flix extends Datalog with lattices and monotone functions, Datafun extends the simply-typed lambda calculus with sets, monotone functions, fixed points, and semilattice types. Additionally, the logic language of Flix is declarative, like Datalog, but Datafun is not. Furthermore, although Datafun does not support user-defined lattices, its type system tracks monotonicity of functions, allowing it to ensure termination of fixed-point computations.

Finally, there have been other multi-paradigm programming languages that are based on logic programming. Mercury [74] is a purely declarative logic language that is typed and supports higher-order functions. Oz [60] combines logic, functional, and object-oriented programming, and Logtalk [59] is an object-oriented logic programming language. These languages extend Prolog, which is Turing-complete, while Flix extends Datalog, which is less expressive but can guarantee termination.

5.3 Implementation Techniques

Many sources examine how functional languages can be implemented. Reynolds [65] considers definitional interpreters for a functional call-by-value language with if-expressions, let-bindings and higher-order functions. However, the interpreters are meant to serve as *definitions* of the functional language, rather than *implementations*, as they sacrifice efficiency for clarity. Cardelli [15] describes a compiler for ML and includes a simple example of a pattern matching implementation. Augustsson [10] presents a compiler for Lazy ML, a lazy and purely functional variant of ML, and provides a brief overview of pattern-matching compilation and lambda lifting.

Peyton Jones [62] provides a comprehensive discussion on implementing a functional language, with specific chapters on pattern matching, efficient compilation of pattern match expressions, and lambda lifting. Appel [5] uses continuation-passing style to implement a compiler for Standard ML and also discusses closure conversion. Finally, in a separate textbook, Appel [6] studies compiler fundamentals, and includes a chapter on functional programming languages, with a section on closure conversion.

The rest of this section explores pattern matching (5.3.1) and lambda functions (5.3.2).

5.3.1 Pattern Matching

Pattern matching is a common feature in functional programming languages, such as Racket [26], Haskell [37], Standard ML [57], F# [77], and Scala [24], and there have been proposals to add pattern matching to C++ [73], Java [35, 52, 66], and Go [44]. In addition to pattern matching in case expressions, as supported by Flix, some languages also allow pattern matching in function definitions. Parameters are specified as patterns and matched against function call arguments, allowing the programmer to define multiple implementations of the same function. Flix does not support this syntax, but the function definitions can easily be transformed into pattern matching in case expressions [10].

The two main approaches for pattern-matching compilation are backtracking automata and decision trees. In general, backtracking generates less but slower code, while decision trees generate faster but more code. Flix takes the former approach. Augustsson [11] provides the earliest description of pattern-matching compilation, using backtracking, while Cardelli [15] uses decision trees. Le Fessant and Maranget [48] discuss optimizations to the backtracking approach, while Maranget [55] studies efficient decision trees. Finally, Maranget [54] and Isradisaikul and Myers [40] propose improving pattern matching usability by producing warnings and performing exhaustiveness checks.

5.3.2 Lambda Functions

While lambda functions are implemented in practically all functional programming languages, they are less common in mainstream object-oriented languages. However, there is a clear trend toward using lambda functions in these languages, as support has been added to C++11 [38, 41] and Java 8 [31, 33].

The term *lambda lifting* was first used by Johnsson [42], who provides an algorithm and an attribute grammar. Since then, there have been many variations and approaches to lambda lifting and closure conversion. Appel and Jim [8] describe a continuation-passing style code generator that includes a closure converter. Minamide et al. [58] present a closure conversion algorithm and a closure representation that is type-directed and type-preserving. Steckler and Wand [76] consider lightweight closure conversion, a situation where some of the free variables of a function are available at all invocation sites, and therefore do not need to be bound in a closure.

One important distinction between Flix and these approaches is that the Flix implementation directly supports closures, and provides a specific data structure for closures. Captured values are implicitly stored within this structure and passed to closure functions as ordinary arguments. In contrast, the related approaches consider the more general case where implementations do not have specific closure support. Captured variables are stored in a general structure like a map or tuple, which is explicitly passed as an argument to closure functions.

There has also been much discussion on optimizing closure implementations. Appel and Jim [7] discuss old and new closure representation strategies and benchmark them for performance. Goldberg and Park [32] describe an escape analysis that determines when closures can be stack-allocated instead of heap-allocated. Shao and Appel [70] present an algorithm for closure conversion that is both efficient and safe for space. Dragoş [20] discusses specific optimizations for the Scala language. Finally, Keep et al. [46] describe a series of optimizations that do not incur any overhead.

Some of these techniques could be applied to Flix, which does not currently optimize its closure implementation. However, the closure representation cannot be changed, since it is provided by the Java standard library—the use of `invokedynamic` allows the closure representation to be defined at run time. On the other hand, any improvements to the Java closure representation will automatically benefit Flix.

Chapter 6

Future Work

The Flix language is still evolving. As new features are added, the implementation will need to be updated. Syntactic changes can be isolated to the front-end, or may require additional AST transformations. Some features may involve only the logic language, leaving the functional language back-ends unchanged. However, changes to the functional language will need to be implemented twice, in both the interpreter and code generator. The hope is that new features can be easily prototyped and tested in the interpreter, before being implemented in the code generator.

The other category of future work is performance, as many improvements can be made to the code generator. For example, if-expressions could be compiled using destination-driven code generation, which reduces code size [21]. The code for calling external functions involves multiple layers of indirection and could be simplified. The implementation of pattern matching could possibly be improved. AST optimizations, such as constant folding, constant propagation, copy propagation, and dead code elimination, could be applied before code generation. Peephole optimizations could also be applied after code generation. Tail call optimization would convert tail recursion to iteration, reducing the stack memory used in recursive functions. Smaller and more efficient data structures would also reduce memory consumption. Furthermore, profiling the code generator and generated bytecode could reveal further opportunities for writing or generating more efficient code.

Finally, although outside the scope of this thesis, improving the implementation of the logic language could lead to significant speedup, since in many Flix programs, most of the execution time is spent in the solver. Currently, the solver evaluates rules by interpreting the AST. Replacing the solver with a back-end that generated bytecode to evaluate the rules should significantly improve performance.

Chapter 7

Conclusions

This thesis presented the implementation of the functional language of Flix, as well as its evaluation. The functional language implementation has two back-ends: an AST interpreter and a JVM bytecode generator. Additionally, the AST produced by the front-end must undergo a number of transformations before it can be consumed by the back-ends. These transformations were also covered in the thesis.

Although the interpreter could directly use the untransformed AST, this is much more difficult, if not impossible, for the code generator. Thus, transformations convert an AST that closely resembles the source program to an AST with lower-level primitives. These transformations include pattern-matching compilation, closure conversion and lambda lifting, and variable numbering.

The AST interpreter was straightforward to implement, though one small challenge was supporting the three different types of function calls: ordinary function calls, closure calls, and external function calls. However, the interpreter is still relatively easy to understand and maintain, which makes it useful for debugging and prototyping new features. In contrast, the code generator is much more complicated, but provides better performance. Implementation challenges included determining how to organize, load, and execute bytecode; reconciling the Flix integer semantics with the JVM integer semantics; and implementing closures with `invokedynamic` and the Java standard library.

This thesis also evaluated the two functional language back-ends in terms of correctness and performance. Over 500 correctness tests were implemented in the ScalaTest framework. Additionally, the Strong Update analysis, a points-to analysis for C programs, was implemented in Flix and compared to a Datalog implementation that ran on the DLV solver. For performance, six benchmarks were written: `fib`, `nbody`, `pidigits`, `matrixmult`,

`shortestpaths`, and `strongupdate`. The first three benchmarks are purely functional programs, and were also used to compare Flix with Ruby, Scala, Java, and C++. `matrixmult` and `shortestpaths` require both functional and logic code, and so only compared the interpreter to the generated bytecode. Finally, `strongupdate` was used as a real-world benchmark, and compared the Flix implementations to Datalog and a handwritten C++ implementation.

The results show that compiled Flix is faster than interpreted Flix. In some cases, such as `fib` and `nbody`, compiled Flix is significantly faster. However, for `matrixmult` and `shortestpaths`, most of the execution time is spent in the solver, evaluating logic code, so the performance of the two functional back-ends is similar. Finally, for `strongupdate`, both the interpreted and compiled Flix implementations are faster than the Datalog implementation, though substantially slower than the handwritten C++ analyzer.

Implementing a bytecode generator for the functional language back-end is the first significant step towards better performance. However, there is still room for improvement, by applying optimization phases or generating more efficient code. Another significant performance improvement would be to implement a code generator for the logic language. As the performance of Flix improves and approaches the performance of handwritten analyzers, Flix will become a more attractive language for implementing static analyses.

References

[1] The Computer Language Benchmarks Game. URL http://benchmarksgame.alioth.debian.org/.

[2] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit Substitutions. *Journal of Functional Programming (JFP)*, 1991. DOI 10.1017/S0956796800000186.

[3] Nicholas Allen, Bernhard Scholz, and Padmanabhan Krishnan. Staged Points-to Analysis for Large Code Bases. In *Proc. Compiler Construction (CC)*, 2015. DOI 10.1007/978-3-662-46663-6_7.

[4] María Alpuente, Marco Antonio Feliú, Christophe Joubert, and Alicia Villanueva. Datalog-Based Program Analysis with BES and RWL. In *Proc. Datalog Reloaded*, 2011. DOI 10.1007/978-3-642-24206-9_1.

[5] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992. URL http://www.cambridge.org/us/knowledge/isbn/item1116671.

[6] Andrew W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 1998. URL http://www.cs.princeton.edu/~appel/modern/java/.

[7] Andrew W. Appel and Trevor Jim. Optimizing Closure Environment Representation. Technical Report TR-168-88, Princeton University, 1988. URL https://www.cs.princeton.edu/research/techreps/TR-168-88.

[8] Andrew W. Appel and Trevor Jim. Continuation-Passing, Closure-Passing Style. In *Proc. Principles of Programming Languages (POPL)*, 1989. DOI 10.1145/75277.75303.

[9] Michael Arntzenius and Neelakantan R. Krishnaswami. Datafun: a Functional Datalog. In *Proc. International Conference on Functional Programming*, 2016. to appear, preprint URL http://www.cs.bham.ac.uk/~krishnan/datafun.pdf.

[10] Lennart Augustsson. A Compiler for Lazy ML. In *Proc. LISP and Functional Programming (LFP)*, 1984. DOI 10.1145/800055.802038.

[11] Lennart Augustsson. Compiling Pattern Matching. In *Proc. Functional Programming Languages and Computer Architecture (FPCA)*, 1985. DOI 10.1007/3-540-15975-4_48.

[12] Avgustinov, Pavel, de Moor, Oege, Jones, Michael Peyton, and Schäfer, Max. QL: Object-oriented Queries on Relational Data. In *Proc. European Conference on Object-Oriented Programming (ECOOP)*, 2016. DOI 10.4230/LIPIcs.ECOOP.2016.2.

[13] Martin Bravenboer and Yannis Smaragdakis. Strictly Declarative Specification of Sophisticated Points-to Analyses. In *Proc. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2009. DOI 10.1145/1640089.1640108.

[14] Eric Bruneton, Romaine Lenglet, and Thierry Coupaye. ASM: a code manipulation tool to implement adaptable systems, 2002. URL http://asm.ow2.org/current/asm-eng.pdf.

[15] Luca Cardelli. Compiling a Functional Language. In *Proc. LISP and Functional Programming (LFP)*, 1984. DOI 10.1145/800055.802037.

[16] Stefano Ceri, Georg Gottlob, and Letizia Tanca. What You Always Wanted to Know About Datalog (And Never Dared to Ask). *Trans. Knowledge and Data Engineering (TKDE)*, 1989. DOI 10.1109/69.43410.

[17] Neil Conway, William R. Marczak, Peter Alvaro, Joseph M. Hellerstein, and David Maier. Logic and Lattices for Distributed Programming. In *Proc. Symposium on Cloud Computing (SoCC)*, 2012. doi: 10.1145/2391229.2391230.

[18] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. Principles of Programming Languages (POPL)*, 1977. DOI 10.1145/512950.512973.

[19] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C – A Software Analysis Perspective. In *Proc. Software Engineering and Formal Methods (SEFM)*, 2012. DOI 10.1007/978-3-642-33826-7_16.

[20] Iulian Dragoş. Optimizing Higher-Order Functions in Scala. In *Proc. Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS)*, 2008. URL https://infoscience.epfl.ch/record/128135.

[21] R. Kent Dybvig, Robert Hieb, and Tom Butler. Destination-Driven Code Generation. Technical Report 302, Indiana University, 1990. URL http://www.cs.indiana.edu/cgi-bin/techreports/TRNNN.cgi?trnum=TR302.

[22] École Polytechnique Fédérale de Lausanne (EPFL). Scala 2.12.0-M2 is now available!, 2015. URL http://scala-lang.org/news/2.12.0-M2.

[23] Michael Eichberg and Ben Hermann. A Software Product Line for Static Analyses: The OPAL Framework. In *Proc. State of the Art in Java Program Analysis (SOAP)*, 2014. DOI 10.1145/2614628.2614630.

[24] Burak Emir, Martin Odersky, and John Williams. Matching Objects with Patterns. In *Proc. European Conference on Object-Oriented Programming (ECOOP)*, 2007. DOI 10.1007/978-3-540-73589-2_14.

[25] Stephen Fink and Julian Dolby. WALA – The T. J. Watson Libraries for Analysis. URL http://wala.sf.net/.

[26] Matthew Flatt and PLT. Racket: Reference. Technical Report PLT-TR-2010-1, PLT Design Inc., 2010. URL https://racket-lang.org/tr1/.

[27] Robert W. Floyd. Algorithm 97: Shortest Path. *Communications of the ACM (CACM)*, 1962. DOI 10.1145/367766.368168.

[28] Free Software Foundation. The GNU Multiple Precision Arithmetic Library. URL https://gmplib.org/.

[29] Miguel Garcia. A new backend and bytecode optimizer for scalac, 2014. URL https://magarciaepfl.github.io/scala/.

[30] Jeremy Gibbons. An Unbounded Spigot Algorithm for the Digits of Pi. *American Mathematical Monthly*, 2006. URL http://www.comlab.ox.ac.uk/oucl/work/jeremy.gibbons/publications/spigot.pdf.

[31] Brian Goetz. Translation of Lambda Expressions, 2012. URL http://cr.openjdk.java.net/~briangoetz/lambda/lambda-translation.html.

[32] Benjamin Goldberg and Young Gil Park. Higher Order Escape Analysis: Optimizing Stack Allocation in Functional Program Implementations. In *Proc. European Symposium on Programming ESOP*, 1990. DOI 10.1007/3-540-52592-0_61.

[33] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification – Java SE 8 Edition*, 2015. URL http://docs.oracle.com/javase/specs/jls/se8/jls8.pdf.

[34] Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. CodeQuest: Scalable Source Code Queries with Datalog. In *Proc. European Conference on Object-Oriented Programming (ECOOP)*, 2006. DOI 10.1007/11785477_2.

[35] Nadeem Abdul Hamid. Pattern Matching on Objects in Java. *Journal of Computing Sciences in Colleges*, 2009. URL http://dl.acm.org/citation.cfm?id=1619221.1619231.

[36] Shan Shan Huang, Todd Jeffrey Green, and Boon Thau Loo. Datalog and Emerging Applications: An Interactive Tutorial. In *Proc. Management of Data (SIGMOD)*, 2011. DOI 10.1145/1989323.1989456.

[37] Paul Hudak, Simon Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph Fasel, María M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Will Partain, and John Peterson. Report on the Programming Language Haskell: A Non-strict, Purely Functional Language. *SIGPLAN Notices*, 1992. DOI 10.1145/130697.130699.

[38] International Organization for Standardization. *ISO International Standard ISO/IEC 14882:2011 Programming Language C++*, 2011. Draft URL http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf.

[39] International Organization for Standardization. *ISO International Standard ISO/IEC 14882:2014 Programming Language C++*, 2014. Draft URL http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4296.pdf.

[40] Chinawat Isradisaikul and Andrew C. Myers. Reconciling Exhaustive Pattern Matching with Objects. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2013. DOI 10.1145/2491956.2462194.

[41] Jaakko Järvi and John Freeman. Lambda Functions for C++0x. In *Proc. Symposium on Applied Computing (SAC)*, 2008. DOI 10.1145/1363686.1363735.

[42] Thomas Johnsson. Lambda Lifting: Transforming Programs to Recursive Equations. In *Proc. Functional Programming Languages and Computer Architecture (FPCA)*, 1985. DOI 10.1007/3-540-15975-4_37.

[43] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. Soufflé: On Synthesis of Program Analyzers. In *Proc. Computer Aided Verification (CAV)*, 2016. DOI 10.1007/978-3-319-41540-6_23.

[44] Chanwit Kaewkasi and Pitchaya Kaewkasi. Pattern Matching for Object-like Structures in the Go Programming Language. In *Proc. Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS)*, 2011. DOI 10.1145/2069172.2069180.

[45] John B. Kam and Jeffrey D. Ullman. Monotone Data Flow Analysis Frameworks. *Acta Informatica*, 1977. DOI 10.1007/BF00290339A.

[46] Andrew W. Keep, Alex Hearn, and R. Kent Dybvig. Optimizing Closures in O(0) Time. In *Proc. Scheme and Functional Programming (Scheme)*, 2012. DOI 10.1145/2661103.2661106.

[47] Monica S. Lam, John Whaley, V. Benjamin Livshits, Michael C. Martin, Dzintars Avots, Michael Carbin, and Christopher Unkel. Context-Sensitive Program Analysis As Database Queries. In *Proc. Principles of Database Systems (PODS)*, 2005. DOI 10.1145/1065167.1065169.

[48] Fabrice Le Fessant and Luc Maranget. Optimizing Pattern Matching. In *Proc. International Conference on Functional Programming (ICFP)*, 2001. DOI 10.1145/507635.507641.

[49] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV System for Knowledge Representation and Reasoning. *ACM Transactions on Computational Logic (TOCL)*, 2006. DOI 10.1145/1149114.1149117.

[50] Ondřej Lhoták and Kwok-Chiang Andrew Chung. Points-To Analysis with Efficient Strong Updates. In *Proc. Principles of Programming Languages (POPL)*, 2011. DOI 10.1145/1926385.1926389.

[51] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification – Java SE 8 Edition*, 2015. URL http://docs.oracle.com/javase/specs/jvms/se8/html/index.html.

[52] Jed Liu and Andrew C. Myers. JMatch: Iterable Abstract Pattern Matching for Java. In *Proc. Practical Aspects of Declarative Languages (PADL)*, 2003. DOI 10.1007/3-540-36388-2_9.

[53] Magnus Madsen, Ming-Ho Yee, and Ondřej Lhoták. From Datalog to Flix: A Declarative Language for Fixed Points on Lattices. In *Proc. Programming Language Design and Implementation (PLDI)*, 2016. DOI 10.1145/2908080.2908096.

[54] Luc Maranget. Warnings for pattern matching. *Journal of Functional Programming (JFP)*, 2007. DOI 10.1017/S0956796807006223.

[55] Luc Maranget. Compiling Pattern Matching to Good Decision Trees. In *Proc. ML*, 2008. DOI 10.1145/1411304.1411311.

[56] Florian Martin. PAG – an efficient program analyzer generator. *Journal on Software Tools for Technology Transfer (STTT)*, 1998. DOI 10.1007/s100090050017.

[57] Robin Milner. A Proposal for Standard ML. In *Proc. LISP and Functional Programming (LFP)*, 1984. DOI 10.1145/800055.802035.

[58] Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed Closure Conversion. In *Proc. Principles of Programming Languages (POPL)*, 1996. DOI 10.1145/237721.237791.

[59] Paulo Moura. *Logtalk: Design of an Object-Oriented Logic Programming Language*. PhD thesis, Universidade da Beira Interior, 2003. URL http://logtalk.org/papers/thesis.pdf.

[60] Martin Müller, Tobias Müller, and Peter Van Roy. Multi-Paradigm Programming in Oz. In *Proc. Visions for the Future of Logic Programming*, 1995. URL http://mozart.github.io/publications/abstracts/Visions95.html.

[61] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An Overview of the Scala Programming Language. Technical Report LAMP-REPORT-2004-006, École Polytechnique Fédérale de Lausanne (EPFL), 2004. URL https://infoscience.epfl.ch/record/52656.

[62] Simon Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987. URL http://research.microsoft.com/en-us/um/people/simonpj/papers/slpj-book-1987/index.htm.

[63] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002. URL http://www.cis.upenn.edu/~bcpierce/tapl/.

[64] Norman Ramsey, João Dias, and Simon Peyton Jones. Hoopl: A Modular, Reusable Library for Dataflow Analysis and Transformation. In *Proc. Haskell Symposium (Haskell)*, 2010. DOI 10.1145/1863523.1863539.

[65] John C. Reynolds. Definitional Interpreters for Higher-Order Programming Languages. In *Proc. ACM Annual Conference*, 1972. DOI 10.1145/800194.805852.

[66] Adam Richard. OOMatch: Pattern Matching as Dispatch in Java. Master's thesis, University of Waterloo, 2007. URL http://hdl.handle.net/10012/3217.

[67] John R. Rose. Bytecodes meet Combinators: invokedynamic on the JVM. In *Proc. Virtual Machines and Intermediate Languages (VMIL)*, 2009. DOI 10.1145/1711506.1711508.

[68] Koichi Sasada. YARV: Yet Another RubyVM: Innovating the Ruby Interpreter. In *Companion to Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2005. DOI 10.1145/1094855.1094912.

[69] Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. On Fast Large-Scale Program Analysis in Datalog. In *Proc. Compiler Construction (CC)*, 2016. DOI 10.1145/2892208.2892226.

[70] Zhong Shao and Andrew W. Appel. Efficient and Safe-for-space Closure Conversion. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2000. DOI 10.1145/345099.345125.

[71] Yannis Smaragdakis and Martin Bravenboer. Using Datalog for Fast and Easy Program Analysis. In *Proc. Datalog Reloaded*, 2011. DOI 10.1007/978-3-642-24206-9_14.

[72] Yannis Smaragdakis, Martin Bravenboer, and Ondřej Lhoták. Pick Your Contexts Well: Understanding Object-Sensitivity. In *Proc. Principles of Programming Languages (POPL)*, 2011. DOI 10.1145/1926385.1926390.

[73] Yuriy Solodkyy, Gabriel Dos Reis, and Bjarne Stroustrup. Open Pattern Matching for C++. In *Proc. Generative Programming: Concepts & Experiences (GPCE)*, 2013. DOI 10.1145/2517208.2517222.

[74] Zoltan Somogyi, Fergus James Henderson, and Thomas Charles Conway. The implementation of Mercury, an efficient purely declarative logic programming language. In *Proc. Australian Computer Science Conference*, 1995. URL http://www.mercurylang.org/documentation/papers/acsc95.ps.gz.

[75] Standard Performance Evaluation Corporation. SPEC Benchmarks. URL https://spec.org/benchmarks.html.

[76] Paul A. Steckler and Mitchell Wand. Lightweight Closure Conversion. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1997. DOI 10.1145/239912.239915.

[77] Don Syme, Gregory Neverov, and James Margetson. Extensible Pattern Matching via a Lightweight Language Extension. In *Proc. International Conference on Functional Programming (ICFP)*, 2007. doi: 10.1145/1291151.1291159.

[78] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java Bytecode Optimization Framework. In *Proc. Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, 1999. URL http://dl.acm.org/citation.cfm?id=781995.782008.

[79] Bill Venners. ScalaTest. URL http://www.scalatest.org/.

[80] John Whaley and Monica S. Lam. Cloning-Based Context-Sensitive Pointer Alias Analysis Using Binary Decision Diagrams. In *Proc. Programming Language Design and Implementation (PLDI)*, 2004. DOI 10.1145/996841.996859.