Eberhard Karls Universität Tübingen
Mathematisch-Naturwissenschaftliche Fakultät
Wilhelm-Schickard-Institut für Informatik

Bachelor Thesis in Media Informatics

Kernel Language to LLVM Compiler

Noah Doersing

February 28, 2017

Reviewer

Prof. Dr. Torsten Grust
Database Systems Research Group
University of Tübingen

Supervisors

Daniel O'Grady & Tobias Müller
Database Systems Research Group
University of Tübingen

Doersing, Noah:
Kernel Language to LLVM Compiler
Bachelor Thesis in Media Informatics
Eberhard Karls Universität Tübingen
Period: November 1, 2016 – February 28, 2017

Abstract

This thesis document aims to give insight into the development of a compiler for a small imperative programming language. This Kernel Language was designed by the research group to aid in provenance analysis of SQL queries. The compiler is written in Haskell, with the relevant data structures having been implemented in C, and uses the llvm-general bindings to emit LLVM IR code. When run, the emitted program creates two log files, based on which data provenance can be inferred.

Besides a discussion of the core technologies and tools used and detailed explanations of several core parts of the implementation, this thesis document contains an analysis of how the compiler stacks up agains a preexisting KL interpreter and what could be done to optimize the performance and memory usage of executables generated from the LLVM IR code emitted by the compiler.

Contents

1 Introduction **1**

1.1 Compilers 1

1.2 Data Provenance 2

1.3 Overview 2

2 Methods and Tooling **3**

2.1 Pipeline 3

2.1.1 Preparation: SQL to Kernel Language 3

2.1.2 Compilation: Kernel Language to LLVM 5

2.1.3 Provenance Derivation: Logging 5

2.2 Kernel Language 5

2.2.1 Description 6

2.2.2 Parser 8

2.2.3 Typechecker 8

2.3 Logging 9

2.4 Haskell 9

2.5 LLVM 10

2.6 llvm-general 12

2.7 C 13

2.7.1 GLib 13

2.7.2 Boehm GC 13

2.8 Related Work 14

2.9 Installation 15

3 Implementation **16**

3.1 Data Structures 16

3.1.1 Common Interface 16

3.1.2 Types 17

3.1.3 String Builtins 18

3.1.4 Example: Simple Dictionary 19

3.1.5 Example: Fast Dictionary 22
3.1.6 Miscellaneous 24
3.2 Garbage Collection 25
3.3 Logging 26
3.3.1 Example 27
3.4 Type Translation 28
3.5 Code Generation Framework 29
3.5.1 Core 29
3.5.2 Utilities 34
3.5.3 Arithmetic, Comparison and Typecasts 36
3.6 Functions 37
3.7 Statements 38
3.7.1 Variable Declaration 38
3.7.2 Variable Assignment 38
3.7.3 Conditional 38
3.7.4 Loops 39
3.7.5 Append and Update 42
3.7.6 Printing and Assertions 43
3.8 Expressions 43
3.8.1 Literals 44
3.8.2 List Operations 45
3.8.3 Dict/Row Operations 45
3.8.4 Miscellaneous 45
3.9 Operators 46
3.10 Miscellaneous 46

4 Discussion **48**
4.1 Performance & Benchmarks 48
4.2 Code Quality 51
4.3 Challenges 51

5 Future Work **53**
5.1 Data Structures 53
5.2 Memory Management 54
5.3 Copy on Write 54
5.4 Interoperability and Scope 55
5.5 Miscellaneous 56

Bibliography **57**

Chapter 1

Introduction

Compilers are everywhere in programming: compilation takes place whenever one representation of a program needs to be transformed or coerced into a different form. This can involve program analysis, optimization passes or insertion of additional actions into the code. Knowledge of how compilers work is an indispensable tool for many programmers.

In the course of this thesis, I have implemented a compiler that receives input in a small imperative programming language (the *Kernel Language*, shortened to KL from now on) and translates it to LLVM IR code, a portable low-level intermediate representation that can be further processed and executed on many platforms. Optionally, the KL to LLVM compiler can instrument the emitted code, meaning that certain runtime data is written to log files.

Before outlining the structure of this thesis, I will briefly discuss compilers in general and data provenance.

1.1 Compilers

Chris Lattner, the original author of the LLVM compiler framework (which I will introduce in more detail in section 2.5), recently gave a high-level overview of what a compiler does in a podcast appearance:

> *"A compiler is a thing that takes the code that a programmer writes and turns it into something the machine can understand. There's lots of different kinds of computers with lots of different kinds of processors. Most programmers don't want to have to think about that or know about that and they want to program and think in a much higher level than what the actual processor can do, and so the compiler's job is to transform what the human wrote to something that the machine can understand."*
>
> — Chris Lattner [LLAS17]

Compilers commonly consist of multiple passes, beginning with lexical analysis and parsing (converting the code into an internal representation) before code generation coupled with semantic analysis takes place. Finally, optimization passes may increase performance without changing the meaning of the program. [Ern08]

Most compilers *lower* their input programs into a representation that can be executed more directly, as described above. A compiler that merely *trans*forms a program into a similar-level language is often called *transpiler*.

1.2 Data Provenance

While this thesis does not directly tie into data provenance derivation, the LLVM IR code emitted by the compiler contains additional logging statements. These logs, created during execution of the compiled program, enable efficient provenance derivation as described in a recent paper. [Mü16]

Data provenance is metadata describing the "origins of a piece of data and the process by which it arrived in a database" [BKWC01] or in the result of a database query. It comes in multiple flavors:

- *Where-provenance* gives clues about the original location of a piece of data. Given a cell in a query result and its where-provenance, it is immediately obvious which input cells have directly contributed to it, even if the result was computed from multiple inputs.
- *Why-provenance* provides the answer to the question of why some data appears in a result, i.e. "which input table cells were inspected to decide about the existence or contents of an output cell". [Mü16]

1.3 Overview

This thesis document is structured as follows:

In chapter 2, I will discuss the tools used within the scope of this thesis. This will include a short dive into instrumentation of generated code, as well as a longer look at the specific technologies employed here, and why they lend themselves to the tasks at hand. In addition, the steps of the pipeline from an SQL query all the way to provenance information will be outlined.

After this setup work, an overview of the implementation will be given in chapter 3, digging deeper where necessary. The Kernel Language to LLVM compiler can be divided into several distinct components: data structures, log writing, code generation backend, and code generation frontend.

With the toolchain and compiler in place, I will reflect on some implementation and tooling choices in chapter 4. This chapter will also contain some benchmarks comparing the performance of my compiler to the KL interpreter, as well as some comparisons of alternative implementations of the relevant data structures.

To finish things off, in chapter 5 I will take a look at what's left to do to bring the compiler to full feature parity with the interpreter. Additionally, some potential improvements to computation speed and memory usage will be discussed.

> *“Compilers on top of compilers. It’s compilers all the way down.”*
>
> — John D. Cook[1]

Chapter 2

Methods and Tooling

In this chapter, I will mainly detail the toolchain I was working with. This includes the toolchain into which the KL to LLVM compiler will be integrated in the context of the data provenance research project, as well as the compiler-internal toolchain and the technologies it is based on.

In between, I will also talk about what goes into the log files created by the programs emitted by the compiler and how they aid data provenance derivation.

2.1 Pipeline

This section aims to give a brief summary of the pipeline that an SQL query has to go through until its data provenance can be derived, with the remainder of this chapter dedicated to exploring its individual components in more detail.

See figure 2.1 on the next page to give yourself an overview of the pipeline. I recommend revisiting the figure after finishing reading this chapter as a kind of summary.

2.1.1 Preparation: SQL to Kernel Language

Given an SQL query, the first step in the pipeline is executing it using PostgreSQL. When properly configured, PostgreSQL writes log files, which can then be parsed by the log parser. The resulting JSON object is then compiled to Kernel Language.[2] At this point, the KL to LLVM compiler takes over.[3]

An example JSON object corresponding to the SQL query `SELECT 1+1` is shown below:[4]

[1] See `https://twitter.com/CompSciFact/status/803420728884334592`.

[2] Log parsing and KL code generation are currently handled by research group-internal Python and Haskell tools, however a rewrite as part of Denis Hirn’s bachelor thesis is in progress.

[3] Before switching to KL as a self-built intermediary language, the research group used a Python-based toolchain for provenance derivation. [Ism15]

[4] With strange indentation to fit everything into as small a space as possible while highlighting the important bits.

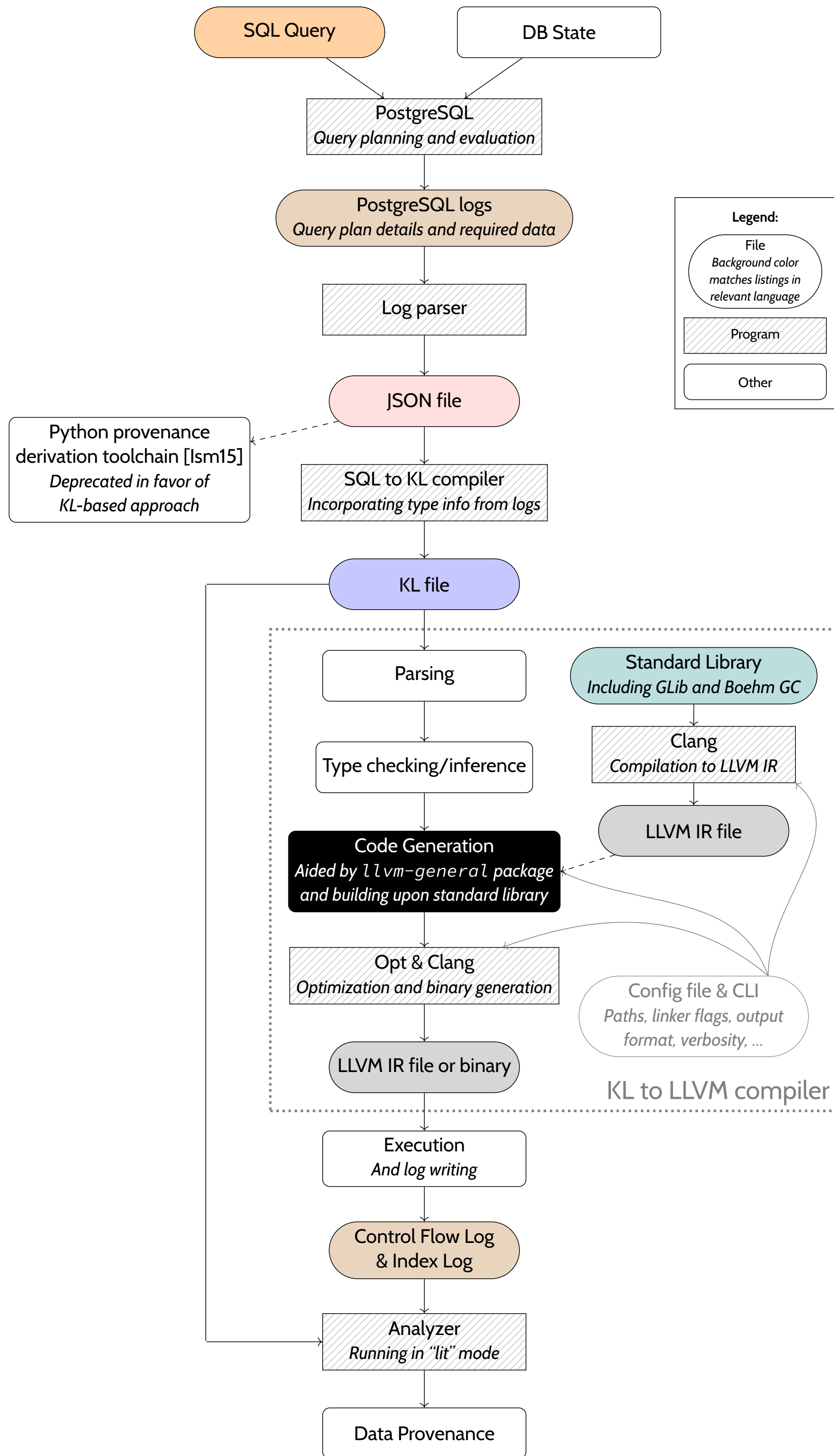

Figure 2.1: Flowchart-like representation of the pipeline.

```json
{"block": {"select": {"agg": false, "distinct": [], "targets": [{
    "expr": {
        "function call": {
            "kind": "pg_operator.+",
            "args": [
                {"const": {
                    "type": {"atom": {"typename": "int4",
                     ↪ "typecategory": "N"}},
                    "value": 1
                }},
                {"const": {
                    "type": {"atom": {"typename": "int4",
                     ↪ "typecategory": "N"}},
                    "value": 1
                }}],
            "type": {"fn": {"args": [{"atom": {"typename": "int4",
             ↪ "typecategory": "N"}}, {"atom": {"typename": "int4",
             ↪ "typecategory": "N"}}], "result": {"atom": {"typename":
             ↪ "int4", "typecategory": "N"}}}},
            "format": "FUNCTION_CALL_OP/BOOL"
        }
    }, "name": "?column?"}], "distinctOn": false}}}
```

2.1.2 Compilation: Kernel Language to LLVM

When running the KL to LLVM compiler, the KL code emitted in the previous step is parsed using the `klparser` package. Then, the `typing` package performs type checking based on inference rules and type annotations in the KL code.[5]

After another minor preprocessing step (grouping of top-level expressions into a function), code generation takes place. This step will be the focus of chapter 3 and can be viewed as a black box transforming the typed KL AST to LLVM IR for now. Once the LLVM IR code corresponding to the input KL file has been generated, the LLVM optimizer takes over and a binary file is generated.[6]

2.1.3 Provenance Derivation: Logging

When run, the binary emitted by the KL to LLVM compiler creates two log files containing control flow decisions and indexes used to access data structures. An example is shown in section 3.3.1. When configured correctly, the `analyzer` executable[7] will automatically read these logs as well as the original KL file and derive the relevant provenance information. [Mü16]

2.2 Kernel Language

The Kernel Language (KL) is a research group-internal language that serves as an intermediary between a source language and whatever target representation is chosen to facilitate provenance derivation.

[5] Both packages are part of the research group-internal KL toolchain and have not been written by me.
[6] Optimization level and output format can be controlled with command-line flags as outlined in section 3.10.
[7] The research group-internal `analyzer` tool was largely written by Tobias Müller.

Currently, the focus lies on SQL as the source language (taking a detour through PostgreSQL, yielding logs which are then transformed to JSON, as previously discussed), but supporting other query languages is merely a matter of writing an appropriate transpiler.

2.2.1 Description

KL is a simple imperative language, providing many of the common imperative constructs, but no global variables or I/O (apart from printing debug messages). The language supports function definitions (and calls), with the notable feature that function arguments are passed by value (i.e. implicit copying). Other operations, such as insertion of values into data structures and updates thereof exhibit the same behavior.

Three types of data structures are available in KL:

- Lists, denoted by `[a, b, ...]`. All elements in a list must have the same type. New elements may be inserted after initialization, but updates are not possible. The `len()` builtin returns the length of a list, while the `contains` infix operator checks whether a list contains a given value.

- Dictionaries ("dicts"), `{k1 : v1, k2 : v2, ...}`. All keys must have the same type and all values must have the same type (which can be different from the key type). Updates of existing values are possible, given the associated key. If the key whose value is to be updated does not already exist, a new key-value pair is inserted. The `keys()` builtin returns a list of the keys of the dict (in undefined order).

- Rows, `<|k1 : v1, k2 : v2, ...|>`. The keys and values can all have different types. Key values and value types must be known on initialization. New key-value pairs cannot be inserted, but the value associated with an existing key may be updated. Mainly used for representing database table contents.

Note that these data types can in principle be nested to any depth – things like a dict mapping a row to a list of rows which themselves map values to lists of rows are possible and actually do occur in translated SQL queries.[8]

In terms of primitive atomic data types, KL supports integers, booleans and floating-point numbers, as well as strings (which are not considered atomic in most other programming languages). A `null` value can be used in place of any value, its type notwithstanding.[9]

Most of the common math and comparison infix operators are available as well.

There are two "flavors" of KL: typed and untyped KL. This is because the pre-exising interpreter[10] requires less type information than the compiler. Type annotations for function arguments, variable declarations and empty literals (`[]` and `{}` for lists and dicts) were added to the language after it became apparent that they would be very helpful in the translation to LLVM IR. Any KL code shown here will be of the typed variant.

[8] See `KLToLLVM/code/examples/Contour_query.kl`, line 447.

[9] This is currently not implemented in the KL to LLVM compiler, see section 5.5.

[10] Which is part of the previously mentioned `analyzer` package.

Example

The piece of KL code at the bottom of this section is a (manual) translation of the SQL query

```sql
SELECT c.captain
  FROM captains c, ships s
 WHERE c.ship = s.ship
   AND s.name = 'Enterprise'
   AND s.launch >= 2300
```

which computes the names of all starfleet captains commanding a ship named "Enterprise" launched after the 23rd century. This involves two selections on the `ships` table and a semi-join of the `captains` table and the result of the selections, as well as a projection on the join result, keeping only the name of each captain. In relational algebra, the query could be expressed as

$$\pi_{\texttt{name}}(\texttt{captains c} \ltimes_{\texttt{c.ship}=\texttt{s.ship}} \sigma_{\texttt{s.name}=\texttt{Enterprise} \wedge \texttt{s.launch} \geq \texttt{2300}}(\texttt{ships s})).$$

```
type R = <|'captain': string|>;
type C = <|'captain': string, 'ship': string|>;
type S = <|'ship': string, 'name': string, 'launch': int|>;

def late_enterprise_captains(cs : [C], ss : [S]) : [R] {
    var res : [R];
    res = [] : [R];

    for c in cs do
        var i : int;
        i = 0;
        var exists : bool;
        exists = false;
        while not(exists) and i < len(ss) do
            var s : S;
            s = ss[i];
            exists = (c.ship == s.ship) and (s.name == 'Enterprise') and
              ↪  (s.launch >= 2300);
            i = i + 1
        od;
        if exists then
            append(res, <|'captain': c.captain|>)
        else
            skip
        fi
    od;
    return res
};

var captains : [C];
captains = [
    <|'captain': 'Kirk',    'ship': 'NCC-1701'|>,
    <|'captain': 'Picard',  'ship': 'NCC-1701-D'|>,
    <|'captain': 'Sisko',   'ship': 'DS9'|>,
    <|'captain': 'Janeway', 'ship': 'NCC-74656'|>,
    <|'captain': 'Archer',  'ship': 'NX-01'|>];

var ships : [S];
```

```
ships = [
    <|'ship': 'DS9',         'name': 'Deep Space 9', 'launch': 2346|>,
    <|'ship': 'NCC-1701',    'name': 'Enterprise',   'launch': 2245|>,
    <|'ship': 'NCC-1701-D', 'name': 'Enterprise',   'launch': 2363|>,
    <|'ship': 'NCC-74656',  'name': 'Voyager',      'launch': 2371|>,
    <|'ship': 'NX-01',       'name': 'Enterprise',   'launch': 2151|>];

var result : [R];
result = late_enterprise_captains(captains, ships);
print(result)
```

The result of the query happens to be the single-element list `[<|'captain': 'Picard'|>]`.

2.2.2 Parser

The KL parser transforms a string containing KL code into an abstract syntax tree (AST). The KL AST is defined in the research group-internal `Kernel` module and shared between various tools related to the data provenance research project. It discerns between types, statements, expressions, literals and builtin operators and provides a distinction between typed and untyped KL.

As an example, this is the part of the untyped AST corresponding to the last line of the KL program from above:

```haskell
Print (Expr (EGet "result"))
```

2.2.3 Typechecker

Given a KL AST with type annotations in the required locations (see the example above), the type checker infers the types of all remaining expressions such as variable access, indexing into data structures and the results of applying an operator to a set of values. If it hits an inconsistency, e.g. trying to subtract a string from an integer, it throws an error (that's the type checking part).

The reason for adding type annotations to KL and performing type inference is that LLVM IR is fairly close to assembly as shown in section 2.5 – as a result, the KL to LLVM compiler needs to know which types it's dealing with in all places.

Again consider the last line of the KL program from section 2.2.1 as an example. After type checking, the `Print` AST node features a type annotation:

```haskell
Print
  (TyExpr
     ( EGet "result"
     , TyList
         (TyRow
            (fromList
               [ ( Right (TyExpr ( EConst (LStr "captain") , TyAtom
                 ↪  TTyStr ))
                 , TyAtom TTyStr
                 )
               ]))
     ))
```

2.3 Logging

As mentioned in the introduction, the compiled KL programs log certain data at runtime. The logged data, which I will explain in detail in this section, enables quasi-static analysis of the KL code, aiding in deriving data provenance.

First, all control flow decisions are logged to a *control flow log*. The control flow is influenced by `if/else` statements: depending on what the associated boolean expression evaluates to, a different block of code is executed next. For this reason, the resulting boolean value is written to the control flow log. Similarly, `while` loops repeatedly evaluate a boolean expression, deciding whether the loop body should be executed or skipped. This boolean is logged in the same way.

The other log file is the *index log*, capturing which parts of data structures have been accessed at which points during the runtime of the program. This simplifies tracing the "path" that a piece of data has taken to get from an input to one or multiple outputs, as well as the computations it was involved in or reshaped by. The following is written to the index log:

- When directly accessing lists, dicts or rows, the indexes/keys are logged.
- List elements cannot be changed after insertion, but when updating values stored in dicts or rows, the keys whose values are modified are logged.
- The return value of the `keys()` builtin is a list of keys, all of which are logged in order.
- The keys appearing in dict and row literals are logged in order as well.
- The indexes used when slicing strings (e.g. with the `substring()` builtin) are not logged. This is because string slicing is not an important use case in the context of the research project and thus does not require fine-grained provenance.

To reduce the size of the log files, some optimizations which will be detailed in section 3.3 have been implemented.

2.4 Haskell

The host language for the implementation is Haskell. This was one of the given requirements and helps integrate the compiler into the existing components of the toolchain as outlined above: the parser, type checker and analyzer are all written in Haskell as well.

The core of Haskell was initially developed in the late 80s and the early 90s. Since then, it has been continually extended, with now-ubiquitous features like `do` notation and list comprehensions added in the late 90s. Major ecosystem improvements like the package systems Cabal and Stack, enabling large-scale development of Haskell software, were added in the last decade. Today, Haskell is widely used in academia, with industry adoption catching up slowly. [HHPJW07]

The most commonly used Haskell compiler is GHC. In addition to supporting the core language, it provides a large number of language extensions [Cha] enabling new features or augmenting existing ones. The Hackage repository contains a large ecosystem of open-source packages which can be installed using Cabal.

Being a functional language, Haskell is well-suited for compiler construction for several reasons:

- Algebraic data types (ADTs) and ASTs are a core part of the language.
- Haskell is strongly and statically typed, and types are inferred automatically where possible. This helps cut down on runtime errors and in my personal experience especially helps catching edge cases.
- It provides sophisticated pattern-matching functionality, which improves readability when dealing with somewhat complicated data types such as the KL AST.
- Idiomatic Haskell code tends to be very succinct – usually, very little boilerplate code is necessary.
- Ideally, a compiler is a large pure function transforming code written in one programming language to another language. In most cases, the target language is lower-level.

  The notion of purity is deeply embedded in Haskell and means that functions have no unforeseen side effects. "For example, if a function `f` has type `\Int -> Int` you can be sure that `f` will not read or write any mutable variables, nor will it perform any input/output. In short, `f` really is a *function* in the mathematical sense: every call `(f 3)` will return the same value." [HHPJW07]

Partly because the LLVM C++ API is called via the FFI (Foreign Function Interface) in the `llvm-general` package (see the next two sections), the KL to LLVM compiler is not strictly pure. This leads to an issue with recurring segmentation faults which will be discussed in section 4.3.

2.5 LLVM

The LLVM project was created and initially maintained by Chris Lattner[11] at the University of Illinois. [LA04] Its core, onto which sub-projects such as Clang, the LLVM debugger and others (as well as the compiler written in the context of this thesis) latch on, "is a modular system for building different kinds of compilers". [LLAS17]

Relevant for this thesis is LLVM's C++ API, which enables automated construction of LLVM IR code, as well as the LLVM optimizer (which runs several optimization passes[12] on the IR) and the associated C/C++/Objective C compiler Clang (which compiles C code to LLVM IR before running optimizations or generating a binary).

LLVM IR is an intermediary representation that is lower-level than C but higher-level than assembly. In the internal pipeline of the KL to LLVM compiler, the IR is passed around in AST form, but it can also be serialized to (and deserialized from) a textual representation.

As an example, consider the following C program:

[11] Lattner was also instrumental in the development of the Clang compiler used here, as well as the Swift programming language.

[12] Custom transformation passes can be supplied as well: `http://www.cs.cornell.edu/~asampson/blog/llvm.html`.

```c
#include <stdio.h>

int main(void) {
    printf("Hello, World!\n");
    return 0;
}
```

Clang[13] transforms it into the following slightly simplified[14] piece of LLVM IR code:

```llvm
; ModuleID = 'helloworld.c'
@.str = private unnamed_addr constant [15 x i8] c"Hello, World!\0A\00",
  ↪  align 1

declare i32 @printf(i8*, ...) #1

define i32 @main() #0 {
entry:
  %retval = alloca i32, align 4
  store i32 0, i32* %retval
  %call = call i32 (i8*, ...)* @printf(i8* getelementptr inbounds ([15 x
    ↪  i8]* @.str, i32 0, i32 0))
  ret i32 0
}
```

Looking at the preceding listing, one can see that in LLVM, everything is explicitly typed: in line 2, a `constant` array (denoted by the brackets) of length `15` and type `i8` (8-bit integer, equivalent to a `char` in C) is created and named `@.str`. `@` denotes globals (such as function definitions, which can only be global), while `%` stands for local values. In line 10, the `getelementpointer` instruction[15] is used to get a pointer of the first element of that array, for which its type is again required, hence `[15 x i8]* @.str` and not simply `@.str`. Similar examples of types being front-and-center can be found in any piece of LLVM IR code.

Something else worth pointing out in the example is the `entry:` label at the top of the `@main()` function. It denotes a *basic block* and can serve as a jump target (LLVM IR defines some conditional and unconditional jump instructions), which is evocative of how code is written in assembly languages.

In stark contrast to many functional languages like Haskell, all functions have to be fully applied when called, and anonymous functions or closures do not exist. A notable difference from object-oriented languages like C++ is that LLVM IR has no concept of classes, making it a fairly simple and very low-level language.

LLVM IR is platform-agnostic when only using basic constructs like in the above example. However, it also provides some intrinsics[16] that may not work on all processor architectures. The KL to LLVM compiler generates portable code that does not rely on any intrinsics.

[13] Which normally emits binaries, but can be set to output unoptimized LLVM IR code with the `-S -emit-llvm -O0` command-line flags.

[14] The full version contains some metadata such as the operating system version, some system architecture information and internal function attributes, all of which are helpful when running optimization passes on the LLVM IR code, but are not relevant in the context of this example.

[15] Documentation available at `http://llvm.org/docs/GetElementPtr.html`.

[16] Functions that map directly to processor instructions. For example, specialized x86 intrinsics might be undefined on ARM and vice versa.

2.6 llvm-general

With Haskell and LLVM locked down as core components of the implementation strategy, what is left to do before writing the KL to LLVM compiler is making them talk to each other.

While LLVM IR code could theoretically be generated directly[17], using a library that ties into the LLVM backend provides a number of advantages. Most notably, the LLVM backend provides a comprehensive catalog of tests and optimization passes, which catch errors and produce faster programs when lowering a Haskell AST into LLVM IR code.

At the time of writing, the most commonly used set of Haskell bindings for LLVM is the `llvm-general` package. It uses the LLVM C++ API to tie itself into LLVM's serialization and deserialization routines, which enables parsing of a file containing LLVM IR code, yielding a AST usable in Haskell code, as well as the lowering of ASTs into LLVM IR code. Also, `llvm-general` provides facilities to utilize the LLVM optimization framework and LLVM's just-in-time execution engine (JIT).

Details of how `llvm-general` is used here will be discussed in the implementation chapter.

It is available for many recent LLVM versions at varying levels of completion: On Hackage, the most recent version at the time of writing[18] is intended for use with LLVM 3.5, while the Github repository[19] also contains branches corresponding to LLVM 3.8 and 3.9. Meanwhile, LLVM 4.0 will be released a few days after the submission deadline of this thesis.

Similar to the examples in the KL section, consider the last line of the example from section 2.2.1. The corresponding two nodes in the `llvm-general` AST created by the KL to LLVM compiler are as follows:[20]

```
1   A.UnName 1200 A.:= A.Load {
2     A.volatile = False,
3     A.address = A.LocalReference A.IntegerType {A.typeBits = 32} (A.UnName
        ↪ 1193),
4     A.maybeAtomicity = Nothing,
5     A.alignment = 0,
6     A.metadata = []
7   },
8   A.UnName 1201 A.:= A.Call {
9     A.tailCallKind = Nothing,
10    A.callingConvention = A.CC.C,
11    A.returnAttributes = [],
12    A.function = Right (A.ConstantOperand (A.C.GlobalReference A.VoidType
        ↪ (A.Name "kltollvm_println"))),
13    A.arguments = [
14      ((A.ConstantOperand A.C.Int { A.C.integerBits = 32, A.C.integerValue
          ↪ = 5 }), []),
15      ((A.LocalReference A.IntegerType { A.typeBits = 32 } (A.UnName
          ↪ 1200)), [])
16    ],
```

[17] Multiple projects attempting this exist at various stages of completion, as listed in this GitHub issue: `https://github.com/llvm-hs/llvm-hs/issues/8`.

[18] See `https://hackage.haskell.org/package/llvm-general-3.5.1.2`.

[19] See `https://github.com/bscarlet/llvm-general`.

[20] This AST is output when the compiler is set to the highest verbosity level.

```
    A.functionAttributes = [],
    A.metadata = []
  }
```

They are later serialized to the following two lines of LLVM IR code:

```llvm
%721 = load i8** %716
call void @kltollvm_println(i32 5, i8* %721)
```

2.7 C

Since it would be cumbersome and error-prone to implement the data structures required by KL (see 2.2.1) directly in LLVM IR, I decided to use C for this purpose, effectively creating a standard library backing the KL to LLVM compiler.

While the data structures could theoretically be implemented in any language that compiles down to LLVM IR,[21] C has a few advantages:

- Among the most widespread programming languages, C is one of the closest to LLVM IR. The upshot of this is that many concepts from the C world map directly to LLVM IR, such as pointers, arrays, structs, and global variables. This simplifies combining the data structures and the LLVM IR code created by the KL to LLVM compiler.
- LLVM ships with the C/C++/Objective C compiler Clang, the two are effectively optimized to work together. The upshot of this is that compiling C to LLVM is extremely stable and can – not least because C is a fast language in general – yield highly optimized LLVM IR code.
- Because C is so widespread, many stable and fast libraries are written in C. See the following subsections for two examples that are currently used by the KL to LLVM standard library.

2.7.1 GLib

As suggested by my supervisors, I implemented efficient dictionaries on top of GLib, which is a bundle of C libraries maintained by the GNOME Project. It provides mature implementations of most common data structures (and various utilities) such as binary trees, linked lists and notably hash tables, which are relevant in the context of dictionaries.

See section 3.1.5 for how GLib was integrated into the KL to LLVM compiler's data structures.

2.7.2 Boehm GC

In low-level environments like C and LLVM, allocated memory needs to be freed manually. This makes things more efficient, but is also one of the principal sources of memory leaks and errors in programs written in such environments.

[21] For example, GHC provides an LLVM backend: https://downloads.haskell.org/~ghc/7.6.3/docs/html/users_guide/code-generators.html.

One solution to this problem (others will be discussed in section 5.2) is garbage collection.[22] In basic terms, a garbage collector periodically scans the heap, marks memory that is not being referenced anymore and then frees it automatically. Implementing this basic concept efficiently and without long pauses continues to be an area of significant research and several slightly different concepts have been developed. [BCR04]

Luckily, there exist "plug-and-play" libraries such as the Boehm-Demers-Weiser conservative garbage collector (Boehm GC), available at `http://www.hboehm.info/gc/`. It has been around since the early 90s, having been continually improved since then, and has been adopted by various open-source projects.[23][24] This is in part due to its simplicity: From an application programmer's point of view, it seamlessly replaces the default malloc/free heap memory interface. [Kor10]

In simple terms, the Boehm GC "assumes that everything is a pointer. This means that it can find false positive references, like an integer which coincidentally has the value of an address in the heap. As a result, some blocks may stay in memory longer than they would with a non-conservative collector."[25]

See section 3.2 for how the Boehm GC was integrated into the KL to LLVM compiler.

2.8 Related Work

Other compilers for imperative languages using the same toolchain (Haskell, LLVM, and `llvm-general`) exist.

For example, Mateusz Machalica has released a compiler for "a Java-like interpreted and compiled language" using a slightly different approach to code generation.[26] A similar project was undertaken by GitHub user dreamycactus,[27] and Andrea Bernardini has written a compiler for "a foolish programming language" on the same toolstack.[28]

Most notably, a tutorial[29] originally written by Chris Lattner to demonstrate how to use LLVM to construct a compiler for a simple language has been adapted by Stephen Diehl for use with `llvm-general`. [Die]

This tutorial was my entry point into the world of Haskell/LLVM compilers. Because the example code is licensed under an open-source license,[30] I based the core of the KL to LLVM compiler on its code generation framework. See section 3.5.1 for details about it, as well as the modifications I made.

[22] A high-level introduction to garbage collection can be found at `https://blog.plan99.net/modern-garbage-collection-911ef4f8bd8e`.

[23] See `http://www.hboehm.info/gc/#users`.

[24] The Webkit project recently rolled out a new garbage collector with the goal of improving JavaScript performance in Webkit-based browsers such as Apple's Safari. The article introducing this change makes several references to the Boehm GC, remarking that its "conservative scan is almost like what the [Boehm GC] collector does". [Piz]

[25] From `http://stackoverflow.com/a/4796449/7504654`.

[26] Available at `https://github.com/stupaq/jvmm`.

[27] See `https://github.com/dreamycactus/j0-compiler`.

[28] See `https://github.com/andrebask/kangaroo`.

[29] See `http://llvm.org/docs/tutorial/`.

[30] MIT license, see `https://github.com/sdiehl/kaleidoscope/blob/master/LICENSE-MIT`.

2.9 Installation

Detailed instructions for setting up the KL to LLVM compiler on OS X are located in `KLToLLVM/code/README.md`. This involves installing Haskell, LLVM, the required C libraries (GLib and the Boehm Garbage Collector) along with their dependencies, the research group-internal tools as well as the compiler itself.

"Talk is cheap. Show me the code."

— Linus Torvalds[1]

Chapter 3

Implementation

In the central chapter of this thesis, I will describe the implementation of the KL to LLVM compiler in a bottom-up manner.

First, I will dive into the C implementation of the data structures, logging and related low-level helper functions. Then, the core of the code generation capabilities will be discussed, before an overview of the code generation for specific language constructs is given, cherry-picking the most interesting bits. To conclude this chapter, I will briefly outline what the remaining modules of the Haskell implementation do.

3.1 Data Structures

For the reasons detailed in section 2.7, C was chosen as the host language for the implementation of the data structures.[2]

3.1.1 Common Interface

As previously outlined, KL features three non-atomic data types: lists, dictionaries and rows. Strings and the standard atomics complete the available types. To support more than very simple KL programs, the compiler needs a way of storing this structured data, as well as accessing it for various purposes like indexing, equality checking, and logging.

For every data type `T`, at least the following functions are implemented:

- `unsigned int kltollvm_T_hash(void*)`: Computes a hash of a value of type `T` (which is passed as a pointer).
- `bool kltollvm_T_equal(void*, void*)`: Checks if two values of type `T` are equal.
- `STR kltollvm_T_tostr(void*)`: Pretty-prints a value into a string, which can then be printed, compared with another string or etc.[3]

[1] See `https://news.ycombinator.com/item?id=902216`.

[2] My `KlToLLVM` repository actually contains more lines of C code than Haskell code – which, more than anything else, speaks to the conciseness of Haskell.

[3] In place of this function, there used to only be a print function. It has been redefined to use the newer `..._tostr` function, which is more versatile: it can also be used to aid unit testing and was used for an initial proof-of-concept approach to logging.

- `void kltollvm_T_log_idx(void*)`: Appends the value to the index log.
- `void *kltollvm_T_deepcopy(void*)`: Creates a copy of the value. If it is a list, dictionary or row, also recursively copies all elements.
- `void kltollvm_T_free(void*)`: Deallocates, i.e. releases the heap memory previously containing the value.

Note that all functions are prefixed with `kltollvm_`. This is because they live in the same namespace as functions defined in the KL code to be compiled, which could lead to conflicts without prefixing.

The above functions only know how to work with their respective data type. More general functions (with an additional type parameter) can be found in `misc.c`, where void pointers are used for a kind of ad hoc polymorphism. These functions switch based on the type and call the more specific function, e.g.:

```c
STR kltollvm_tostr(TYPE t, void *vv) {
    if (vv == 0) {
        return kltollvm_str_deepcopy("!!!NULLPTR!!!");
    }
    switch (t) {
        case TTYNULL:
            return kltollvm_null_tostr(vv);
        case TTYBOOL:
            return kltollvm_bool_tostr(vv);
        case TTYINT:
            return kltollvm_int_tostr(vv);
        case TTYFLOAT:
            return kltollvm_float_tostr(vv);
        case TTYSTR:
            return kltollvm_str_tostr(vv);
        case TYLIST:
            return kltollvm_list_tostr(vv);
        case TYDICT:
            return kltollvm_dict_tostr(vv);
        case TYROW:
            return kltollvm_row_tostr(vv);
        default:
            return "";
    }
}
```

3.1.2 Types

The type parameter in the above function (and in many other places in the C codebase) is of type `TYPE`, which is defined as follows:[4]

```c
typedef int TYPE;
const TYPE TTYNULL  = 0;
const TYPE TTYBOOL  = 1;
const TYPE TTYINT   = 2;
const TYPE TTYFLOAT = 3;
const TYPE TTYSTR   = 4;
```

[4]Note that this needs to match the type translation functions in the Haskell `Codegen.Type` module described in section 3.4.

```
const TYPE TYLIST   = 5;
const TYPE TYDICT   = 6;
const TYPE TYROW    = 7;
```

The naming is inspired by the `Kernel` module defining the AST. Complex data types like `TYLIST`, `TYDICT` and `TYROW` store the types of their elements internally, removing the need of keeping track of them at the container type level.

3.1.3 String Builtins

KL provides some built-in functions that operate on strings, such as `strlen`, `strleft` and `substring`, among others. Additionally, lexicographic comparison is defined on strings. Note that all string builtins are designed to match the behavior of the PostgreSQL functions of the same name.

In this section, I will detail the implementation of the `strleft` and `substring` operators, which can be found in `KLToLLVM/code/cbits/str.c`.

The `strleft` builtin, given an integer `n`, returns the first (i.e. left-most) `n` characters of the string. If `n` is negative, all but the last `|n|` characters are returned. The implementation computes the relevant indexes and calls the `kltollvm_str_substring()` function to do the dirty work:

```c
void* kltollvm_str_strleft(void *vs, int n) {
    STR s = (STR) vs;
    if (n < 0) {
        n = kltollvm_str_strlen(s) - abs(n);
    }
    return kltollvm_str_substring(s, 1, n + 1);
}

void* kltollvm_str_substring(void *vs, int from, int to) {
    STR s = (STR) vs;

    // match 1-indexing found in postgres
    from = from - 1;
    to = to - 1;

    // length of substring
    int l = to - from;

    // allocate new string of correct length
    STR sub = (STR) kltollvm_alloc((l + 1) * sizeof(char));

    // iteratively copy relevant substring
    int i = 0;
    while (i < l) {
        sub[i] = s[i + from];
        i++;
    }

    // add terminating null byte
    sub[i] = '\0';

    return (void*) sub;
}
```

3.1.4 Example: Simple Dictionary

In this section, I will go through my initial dictionary implementation (see KLToLLVM/code/cbits/dict.h and KLToLLVM/code/cbits/dict.c) and explain what each function does (and why it's implemented that way). For brevity and because they are fairly similar and not the focus of this thesis, I won't go into the details of the list and row implementations.

First taking a look at the header file, we can see how the relevant structs, which hold the data contained in a dict, are defined:

```c
typedef struct dictitem DICTITEM;
struct dictitem {
    void *key;
    void *value;
    DICTITEM *next;
};

typedef struct dict DICT;
struct dict {
    int size;
    TYPE key_type;
    TYPE value_type;
    DICTITEM *first;
    DICTITEM *last;
};
```

A dict struct contains size and type data as well as pointers to the first and last dictionary element. In this simple implementation, a dict is basically implemented as a linked list of key-value tuples. This is also apparent in the definition of the dictitem struct, which contains pointers to the key, value and the next element.

When a dict is initialized, the kltollvm_dict_empty() function is called with the desired key and value type, followed by a series of kltollvm_dict_update() calls (which insert a new key-value pair at the end of the dictionary if the key cannot be found). These two functions are implemented as follows:

```c
void *kltollvm_dict_empty(TYPE kt, TYPE vt) {
    DICT *d = (DICT*) kltollvm_alloc(sizeof(DICT));
    d->size = 0;
    d->key_type = kt;
    d->value_type = vt;
    return (void*) d;
}

void kltollvm_dict_update(void *dv, void *kv, void *vv) {
    DICT *d = (DICT*) dv;

    // create a copy of the new value to simulate pass-by-value
    vv = kltollvm_deepcopy(d->value_type, vv);

    // check if key already exists, update if to does
    if (d->size != 0) {
        DICTITEM *i = d->first;
        for (size_t n = 0; n < d->size; n++) {

            // perform deep comparison, assuming types are equal
```

```
21              if (kltollvm_equal(d->key_type, kv, i->key)) {
22
23                  // update value
24                  kltollvm_free(d->value_type, i->value);
25                  i->value = vv;
26
27                  // we're done here (unless the dict contains two entries
                     ↪  with same key, which shouldn't happen)
28                  return;
29              }
30
31              i = i->next;
32          }
33      }
34
35      // guess it doesn't, so let's insert it. first create a copy of the
         ↪  new key (value was copied already) to simulate pass-by-value
36      kv = kltollvm_deepcopy(d->key_type, kv);
37      kltollvm_dict_insert(dv, kv, vv);
38  }
```

Note that in the `kltollvm_dict_update` function, the new value (as well as the key if it does not already exist in the dictionary) is copied using the `kltollvm_deepcopy` function. This is not always necessary, e.g. when the new value is a literal.[5]

When querying a dict for the value matching a given key, then `kltollvm_dict_elemd()` function is called. It traverses the linked list of dict elements and returns a copy of the value or a null pointer if the key could not be found:

```
1   void *kltollvm_dict_elemd(void *dv, void *kv) {                        C
2       DICT *d = (DICT*) dv;
3       DICTITEM *i = d->first;
4
5       for (size_t n = 0; n < d->size; n++) {
6
7           // perform deep comparison and return if match was found
8           if (kltollvm_equal(d->key_type, kv, i->key)) {
9               return kltollvm_deepcopy(d->value_type, i->value);
10          }
11
12          i = i->next;
13      }
14
15      // key not found => null pointer
16      return NULL;
17  }
```

Because all keys of a dictionary (as opposed to a row) are of the same type, they can be returned as a list, e.g. to help iterating through a dictionary. The `kltollvm_dict_keys()` function takes care of that:

```
1   void *kltollvm_dict_keys(void *dv) {                                   C
2       DICT *d = (DICT*) dv;
3
4       // create empty list of dict key type
5       void *l = kltollvm_list_empty(d->key_type);
```

[5]See section 5.3 for a potential fix.

```
6
7       // an empty dictionary has no keys, so let's simply return the
          ↪  previously created empty list
8       if (d->size == 0) {
9           return l;
10      }
11
12      // go through dict, adding keys to our list
13      DICTITEM *i = (DICTITEM*) d->first;
14      for (size_t n = 0; n < d->size; n++) {
15
16          // no need to deepcopy the key as append will take care of that
17          kltollvm_list_append(l, i->key);
18          i = i->next;
19      }
20
21      return (void*) l;
22  }
```

With the dict-specific functions out of the way, I'll go over how the common interface for all data types (see section 3.1.1) is implemented for dictionaries. For brevity's sake, I'll only detail equality checking and logging – the other functions looks somewhat similar.

Given two dictionaries v and w, the implementation first compares their key and value types as well as their sizes. If any mismatch occurs here, it can be inferred that the dictionaries are not equal without looking at any of their elements. Then, the function checks whether v is of size 0.[6] If it is, the two dictionaries are indeed equal. If it isn't, the function iterates through v, checking for each key-value pair if the key exists in w. If it doesn't, the dictionaries are not equal. Otherwise, the values are compared. If after iterating through the first dictionary no mismatch was found, it follows that both dictionaries are equal (since the sizes have been compared earlier):

```c
1   bool kltollvm_dict_equal(void *vv, void *wv) {
2       DICT *v = (DICT*) vv;
3       DICT *w = (DICT*) wv;
4
5       // compare types and sizes
6       if (v->key_type != w->key_type || v->value_type != w->value_type ||
          ↪  v->size != w->size) {
7           return false;
8       }
9
10      // empty dicts are equal (we've already established that both have
          ↪  the same types and size, so we only need to check one)
11      if (v->size == 0) {
12          return true;
13      }
14
15      // for each element in the first dict
16      DICTITEM *vi = v->first;
17      for (size_t n = 0; n < v->size; n++) {
18
19          // try to get corresponding value in other dict
20          void *wi_value = kltollvm_dict_elemd(w, vi->key);
```

[6] Because it's already been established that both dictionaries have the same size, only one of them needs to be checked here.

```
21
22          if (wi_value == NULL) {
23
24              // key doesn't exist in other dict
25              return false;
26          } else if (!kltollvm_equal(v->value_type, vi->value, wi_value)) {
27
28              // keys equal, but values not equal
29              return false;
30          }
31          vi = vi->next;
32      }
33
34      // if we haven't returned anything at this point, they're equal
35      return true;
36  }
```

The function `kltollvm_dict_log_idx()` is called whenever a dictionary appears as part of an index, e.g. when doing an insertion in a row mapping dictionaries to integers. Writing to the global file pointer `kltollvm_fpidx` (which is initialized by the `kltollvm_init_logging()` function, see section 3.3), it serializes the dictionary based on the format expected by the analyzer. First, the string `"VDict (fromList ["` is logged, before iteration through the keys and values triggers logging keys-value pairs as comma-separated tuples. Finally, the log entry is closed by `"])"`.

```c
1   void kltollvm_dict_log_idx(void* dv) {
2       DICT *d = (DICT*) dv;
3       fprintf(kltollvm_fpidx, "VDict (fromList [");
4       if (d->size != 0) {
5           DICTITEM *i = d->first;
6           for (size_t n = 0; n < d->size; n++) {
7               fprintf(kltollvm_fpidx, "(");
8               kltollvm_log_idx(d->key_type, i->key);
9               fprintf(kltollvm_fpidx, ",");
10              kltollvm_log_idx(d->value_type, i->value);
11              fprintf(kltollvm_fpidx, ")");
12
13              // if end hasn't been reached yet, print comma and load next
14              if (n != d->size - 1) {
15                  fprintf(kltollvm_fpidx, ",");
16                  i = i->next;
17              }
18          }
19      }
20      fprintf(kltollvm_fpidx, "])");
21  }
```

3.1.5 Example: Fast Dictionary

An obvious drawback of the dictionary implementation detailed in the previous section is performance: lookups and updates, which dominate the workload in most KL programs, run in $\mathcal{O}(n)$ where n denotes the number of elements in the dictionary.

Efficient dictionary implementations are commonly based on hash tables, where keys are hashed on insert and lookup, reducing the cost to an amortized $\mathcal{O}(1)$.[7]

[7] See `http://www.cs.cornell.edu/courses/cs312/2008sp/lectures/lec20.html`.

As suggested by my supervisors, I used *GHashTable*[8] for this purpose, which is part of GLib (see section 2.7.1). The relevant part of its API consists of the following functions:

- `GHashTable *g_hash_table_new_full (GHashFunc hash_func, GEqualFunc key_equal_func, GDestroyNotify key_destroy_func, GDestroyNotify value_destroy_func)`, where the first and second arguments are ordinary function pointers to the hash and equality checking functions for the key data type, respectively. The last two arguments are function pointers to free functions for keys and values, respectively. This function sets up a hash table and returns a pointer to it.
- `gboolean g_hash_table_replace (GHashTable *hash_table, gpointer key, gpointer value)` updates a key-value pair, inserting it if the key is not already part of the hash table. Note that it replaces the old key with the new one and calls the free function on the now discarded old key.
- `gboolean g_hash_table_lookup (GHashTable *hash_table, gconstpointer key)` returns the value associated with the `key` or a null pointer if the key is not found.
- `void g_hash_table_iter_init (GHashTableIter *iter, GHashTable *hash_table)` initializes the given iterator and `gboolean g_hash_table_iter_next (GHashTableIter *iter, gpointer *key, gpointer *value)` advances the iterator, storing pointers to the current key and value in the given memory locations. See below for an example of how iterators are used.
- `guint g_hash_table_size (GHashTable *hash_table)` returns the size of the hash table as an unsigned integer.
- `void g_hash_table_unref (GHashTable *hash_table)` decrements the reference count of the hash table by 1. If it drops to 0 during this process, all keys and values as well as the hash table itself is freed.

Note that the GLib-defined types are equivalent to the usual C types for all intents and purposes: `gboolean` corresponds to `bool`, `guint` is an `unsigned int` and `gconstpointer` can be written as `const void*`, for instance.[9]

As an example, let's again consider logging. Using the iterator functions introduced above, the `kltollvm_dict_log_idx()` function can be rewritten as follows, where `d->ht` is a GHashTable:[10].

```c
void kltollvm_dict_log_idx(void* dv) {
    DICT *d = (DICT*) dv;

    fprintf(kltollvm_fpidx, "VDict (fromList [");

    GHashTableIter iter;
```

[8]Documentation available at `https://developer.gnome.org/glib/stable/glib-Hash-Tables.html`.

[9]This is documented at `https://developer.gnome.org/glib/stable/glib-Basic-Types.html`.

[10]This improved dictionary implementation can be found at `KLToLLVM/code/cbits/hashdict.c`

```
7       g_hash_table_iter_init(&iter, d->ht);
8       void *key;
9       void *value;
10      while (g_hash_table_iter_next(&iter, &key, &value)) {
11          fprintf(kltollvm_fpidx, "(");
12          kltollvm_log_idx(d->key_type, key);
13          fprintf(kltollvm_fpidx, ",");
14          kltollvm_log_idx(d->value_type, value);
15          fprintf(kltollvm_fpidx, "),");
16      }
17
18      // prepare to overwrite final comma
19      kltollvm_log_idx_seek(-1);
20
21      fprintf(kltollvm_fpidx, "])");
22  }
```

In addition to the speedier dictionary implementation discussed above, there also exists an alternative row implementation based on GLib. It is a bit more complex because it needs to store pairs of type and value due to rown being inhomogenous with regard to key and value types.

Also available are two different list implementations based on a linked list (where each list element contains a pointer to its successor) and an ArrayList/Vector (where list elements are stored in an array which is grown by a factor of 1.5 whenever the allocated space is exhausted) approach, respectively.[11] I will explore how these different implementations stack up against each other in section 4.1.

3.1.6 Miscellaneous

Helper Functions for Loops

Both list implementations contain some helper functions that simplify iterating through a list, which mainly occurs in KL's `for` loops. While not necessary in the arraylist implementation (because direct indexing into the underlying array is possible in $\mathcal{O}(1)$ time there, as opposed to $\mathcal{O}(n)$ for linked lists), the functions are present in both implementations to facilitate easy switching among them as outlined in 3.1.6.

Math Builtins

KL defines four builtin floating-point math operators, three of which correspond to functionality provided in the `math.h`. My standard library basically re-exports these functions. The fourth, `roundto(f, i)`, which rounds `f` to `i` decimal places, is implemented as follows:

```c
1  float kltollvm_roundto(float input, int digits) {
2      return roundf(input * powf(10, digits)) / powf(10, digits);
3  }
```

[11]See `KLToLLVM/code/cbits/linkedlist.c` and `KLToLLVM/code/cbits/arraylist.c`, respectively.

stdlib.c

Bringing things together, all header and source files are imported in `KLToLLVM/code/cbits/stdlib.c`. This file also contains some preprocessor directives that enable quick switching between different data structure implementations, as well as toggling the garbage collector on or off. In addition, this file contains some wrapper functions related to memory management: a memory allocator (see below), reallocator and deallocator.

```c
void *kltollvm_alloc(size_t size) {
    #ifdef USE_GC
        void *mem = GC_MALLOC(size);
    #else
        void *mem = calloc(1, size);
    #endif

    // see https://linux.die.net/man/3/calloc
    if (mem == NULL) {
        printf("Error: Out of memory (while trying to allocate %zu
         ↪  bytes)", size);
        kltollvm_exit_failure();
    }
    return mem;
}
```

Concluding this section, a few words about how the standard library is "combined" with the LLVM IR code generated from KL are in order.

When the KL to LLVM compiler runs, it executes a shell script checking if the standard library has been modified since it was last compiled to LLVM IR. If so (or if it was never compiled before), it is recompiled using Clang (see figure 2.1 in section 2.1). As part of this process, a test suite[12] is run, verifying that the data structures work as intended. The resulting file `KLToLLVM/code/temp/stdlib.ll` is then read by the KL to LLVM compiler and the contained LLVM module is extended with the LLVM IR code generated from the input KL file.

The reason for extending the LLVM IR file (as opposed to linking with it, as is perhaps more common) is that this way, any changes to the data structures are automatically adopted without any further work. The alternative would be to essentially declare all C functions (or at least those that could be called in the generated LLVM IR code) in the code generator. This would duplicate their signatures and thus make changes more cumbersome.[13]

3.2 Garbage Collection

As introduced in section 2.7.2, the Boehm GC is used to patch over memory leaks. Hooking it into the standard library proved to be simple, requiring the following code,[14] with the `kltollvm_init()` function having to be called before any memory allocation (using `GC_MALLOC()` or `GC_REALLOC()` instead of `malloc()` or `realloc()`) takes place:

[12] Located at `KLToLLVM/code/cbits/test.c`.

[13] However, once the standard library is stable and the function signatures are unlikely to change, the linking approach might be favorable.

[14] In addition to passing the correct linker flags during build.

```c
#include <gc.h>

void kltollvm_init() {
    GC_INIT();
}
```

Calls of the `free()` function can be removed after enabling the garbage collector. Interestingly, enabling the garbage collector actually increases performance in some cases, which I will analyze in section 4.1.

A significant downside is that GLib currently does not provide a way to override the functions it uses for memory management, apparently rendering it impossible to use the Boehm GC while using GLib. Up until mid-2015, this was possible by modifying the memory vtable as shown in the example below, but other changes broke this functionality and it was subsequently removed.[15]

```c
g_mem_gc_friendly = TRUE;

GMemVTable memvtable = {
    .malloc      = GC_MALLOC,
    .realloc     = GC_REALLOC,
    .free        = NULL,
    .calloc      = NULL,
    .try_malloc  = NULL,
    .try_realloc = NULL
};

g_mem_set_vtable(&memvtable);
```

3.3 Logging

This thesis would be moot in the context of provenance derivation without capturing the control flow and indexes as described in section 2.3. This is achived using the C functions declared in `KLToLLVM/code/cbits/logging.h` and implemented in `KLToLLVM/code/cbits/logging.c`, as well as the previously mentioned `kltollvm_T_log_idx()` functions.

The logging format is dictated by the analyzer and corresponds to the auto-generated `Show` instance of the `Value` type, defined in `ProvenanceHaskell/analyzer/src/Values.hs`. For example, when logging a row with a single key `"shot first"` and associated value `"han"`, the log entry `VRow $ fromList [("shot first", "han")]` is appended to the index log.

Central to the implementation are the following functions:

- `void kltollvm_init_logging()`, which opens the log files for writing and writes an inital opening bracket.
- `void kltollvm_finish_logging()`, which makes sure that the log files are correctly terminated with a closing bracket and the file handles are closed.
- `void kltollvm_log_cf(bool b)`, which appends `True` or `False` and a comma to the control flow log.

[15] See https://bugzilla.gnome.org/show_bug.cgi?id=751592.

- `void kltollvm_log_idx_entry(TYPE t, void *idx)`, which logs an appropriate representation of the arguments to the index log, followed by a comma.

These four functions have counterparts in the `Codegen.Util` module (the rest of which will be discussed in section 3.5.2):[16]

```
1  -- | Open the logfiles and initialize them.
2  initLogging :: Codegen Operand
3  initLogging = callCbits T.void "kltollvm_init_logging" []
4
5  -- | Finalize the logs and close the logfiles.
6  finishLogging :: Codegen Operand
7  finishLogging = callCbits T.void "kltollvm_finish_logging" []
8
9  -- | Write a boolean to the control flot logfile.
10 lcput :: Logging → Operand → Codegen Operand
11 lcput l a = when l $ callCbits T.void "kltollvm_log_cf" [a]
12
13 -- | Write an index to the index logfile. If the index is constant and
     ↪ atomic in the KL sense (where strings are atoms too), skip it. The
     ↪ 'Left' case of the third argument should only be used when it's
     ↪ certain that the index is not constant: right now, it's used for
     ↪ logging the list elements returned by the @keys()@ function in KL.
14 liput :: Logging → Operand → Either K.TyValue K.TyExpr → Codegen
     ↪ Operand
15 liput l o (Left ty) = when l $ do
16     o' ← toVoidPtr ty o
17     callCbits T.void "kltollvm_log_idx_entry" [(toOperand . tyValueToC)
         ↪ ty, o']
18 liput l o (Right e) = unless (isAtomicConstant e) $ liput l o $ (Left .
     ↪ K.typeOf) e
19   where
20     isAtomicConstant e = case e of
21         K.TNull{}    → True
22         K.TBool{}    → True
23         K.TInt{}     → True
24         K.TFloat{}   → True
25         K.TStr{}     → True
26         _            → False
```

As can be seen in line 18 (highlighted), atomic constants are not written to the index log. In addition, index accesses in `print` or `assert` statements are ignored as well (see section 3.7.6). Apart from these exceptions, any control flow decisions (occuring in `if`/`else` and `while` statements) are logged, as well as element access (plus the list returned by the `keys()` built-in, due to dictionary entries being stored in arbitrary order).

3.3.1 Example

When running the example given in section 2.2.1, the following control flow and index logs are written, respectively.

[16] Note that the `when` function in line 11 corresponds to an `if ... then ... else ...` construct without the `else` part, emitting a `noop` instruction instead if the condition evaluates to false.

```
[True,True,True,True,True,False,False,True,True,True,False,True,True,True,True,True,True,False,False,True,True,True,True,True,False,False,True,True,True,True,True,False,False]
```

Log File

```
[VInt 0,VInt 1,VInt 2,VInt 3,VInt 4,VInt 0,VInt 1,VInt 2,VInt 0,VInt 1,VInt 2,VInt 3,VInt 4,VInt 0,VInt 1,VInt 2,VInt 3,VInt 4,VInt 0,VInt 1,VInt 2,VInt 3,VInt 4]
```

Log File

3.4 Type Translation

After discussing the standard library above and before diving into the code generation framework, a brief discussion of how the type information available in the KL AST (after type checking) is translated to LLVM types and constants for use with the standard library.

The `Codegen.Type` module manages any type-related definitions and provides conversion functions. To this end, it reads information about the platform-dependent size of various fundamental data types[17] from a config file (which is explained in more detail in section 3.10). Based on this size data, the atomic data types are defined.

For example, for integers, the configured size is retrieved using the `unsafeIntSize` function before the LLVM integer type is defined:

```haskell
intBits :: Integer
intBits = unsafeIntSize * 8

int :: T.Type
int = T.IntegerType (fromIntegral intBits)
```

With the LLVM types defined in this manner, KL types can be translated to them and the type constants used throughout the standard library (see section 3.1.2):

```haskell
tyValueToLlvm :: K.TyValue -> T.Type
tyValueToLlvm K.TyNull       = none
tyValueToLlvm K.TyBool       = bool
tyValueToLlvm K.TyInt        = int
tyValueToLlvm K.TyFloat      = float
tyValueToLlvm K.TyStr        = string
tyValueToLlvm (K.TyList _)   = list
tyValueToLlvm (K.TyDict _ _) = dict
tyValueToLlvm (K.TyRow  _)   = row
tyValueToLlvm ty             = error $ "function tyValueToLlvm doesn't know the type " ++ show ty

tyValueToC :: K.TyValue -> Int
tyValueToC K.TyNull       = 0
tyValueToC K.TyBool       = 1
tyValueToC K.TyInt        = 2
tyValueToC K.TyFloat      = 3
tyValueToC K.TyStr        = 4
tyValueToC (K.TyList _)   = 5
tyValueToC (K.TyDict _ _) = 6
```

[17] For example, integers are usually 32 bits wide on a 32-bit system and 64 bits wide on a 64-bit system.

```
20  tyValueToC (K.TyRow  _)    = 7
21  tyValueToC a               = error $ "function tyValueToC doesn't know
  ↪  the type " ++ show a
```

Note that the KL parser reads integers into `Integers`. This is an arbitrary-precision integer type, an equivalent of which is not readily available in LLVM. As a result, the KL to LLVM compiler uses the standard integer size on the platform (usually 32 bits). Similarly, strings are implemented as arrays of `i8`, which means that each character is only 8 bits wide, cutting off any unicode code points beyond U+00FF. This could be mitigated as detailed in section 4.2.

3.5 Code Generation Framework

Before talking about how individual language constructs are translated from KL to LLVM IR, I'd like to outline the monadic framework that sits between the `llvm-general` internals and the code generator. As mentioned in section 2.8, this is heavily based on Stephen Diehl's tutorial code, with minor extensions and some additional convenience functions.

Because of this, the framework is fairly extensible and not specific to KL: It could be used to help compile any suitable AST to LLVM IR without major changes.

3.5.1 Core

The `Codegen.Core` module contains the basic functionality of the framework.

First, we will consider the module-level code. For our purposes, an LLVM module corresponds to a file of LLVM IR code, with the module containing all definitions (i.e. functions or constants) that will end up in the output file. The `LLVM` state monad provides a means of adding such definitions to a module.

```
1   newtype LLVM a = LLVM
2       { unLLVM :: State AST.Module a
3       } deriving (Functor, Applicative, Monad, MonadState AST.Module)
4
5   runLLVM :: AST.Module → LLVM a → AST.Module
6   runLLVM = flip (execState . unLLVM)
7
8   emptyModule :: String → AST.Module
9   emptyModule label = defaultModule { moduleName = label }
10
11  -- | Adds a definition to the current module.
12  addDefn :: Definition → LLVM ()
13  addDefn d = do
14      defs ← gets moduleDefinitions
15      modify $ \s → s { moduleDefinitions = defs ++ [d] }
```

Note that functions not described in detail here, e.g. `defaultModule` or `moduleDefinitions` above, are part of the `llvm-general` package. Hopefully, their purpose is obvious from their names and context (or at least clear enough for the code reproduced here to make sense), otherwise please consult the documentation.[18] Using

[18] Available at `https://hackage.haskell.org/package/llvm-general-3.5.1.2#modules`.

an IDE[19] has helped me a lot here.

Building on this, the `define` function adds a function to the module:

```
define :: Type             -- ^ Return type
       → Funcname          -- ^ Name
       → [(Type, Name)]    -- ^ Function signature
       → [BasicBlock]      -- ^ Body
       → LLVM ()
define retty label argtys body = addDefn $
    GlobalDefinition $ functionDefaults
    { name        = Name label
    , parameters  = ([Parameter ty nm [] | (ty, nm) ← argtys], False)
    , returnType  = retty
    , basicBlocks = body
    }
```

With the ability to add function definitions to modules, we'd like to also fill those functions with logic. As visible in the type of the `define` function, function bodies are represented as lists of basic blocks[20] which in turn contain instructions. As a result, we need data types allowing us to assemble basic blocks and prepare them for insertion into a function body:

```
type Varname = String

-- | Maintains a mapping between variable name and current value. The
  ↪ associated `K.TyValue` can be saved as well.
type SymbolTable = [(Varname, (Maybe K.TyValue, Operand))]

-- | Mapping from block names to unique identifiers.
type Names = Map.Map String Int
instance IsString Name where
    fromString = Name . fromString

data CodegenState = CodegenState
    { currentBlock :: Name                       -- ^ Name of the active
                                                 -- block to append to
    , blocks       :: Map.Map Name BlockState    -- ^ Blocks for function
    , symtab       :: SymbolTable                -- ^ Function scope
                                                 -- symbol table
    , blockCount   :: Int                        -- ^ Count of basic blocks
    , count        :: Word                       -- ^ Count of unnamed
                                                 -- instructions
    , names        :: Names                      -- ^ Block Name Supply
    } deriving Show

data BlockState = BlockState
    { idx   :: Int                              -- ^ Block index
    , stack :: [Named Instruction]              -- ^ Stack of instructions
    , term  :: Maybe (Named Terminator)         -- ^ Block terminator
    } deriving Show
```

A `SymbolTable` associates a variable name with the corresponding operand. Optionally, if the variable is generated directly from KL, its KL type can also be stored

[19] My IDE of choice for this project was Atom with the IDE-Haskell package, which has worked out really well apart from occasional slow response times: `https://atom.io/packages/ide-haskell`

[20] See the example in section 2.5 for how these blocks are specified in LLVM IR code.

here. This comes in handy when handling rows and dictionaries because the KL `update(variable, key, value)` statement is parsed and typechecked in such a way that the KL AST contains no type information for the variable to be updated. This means that the code generator wouldn't know whether to perform a dict or row update without saving the type in the symbol table.

The two record types are self-explanatory based on the comments and the following functions.

The `uniqueName` function makes sure that no duplicate block names can appear (if they did, it would not be clear which one to jump to if the block name is set as the target of a jump instruction) by appending the number of previous occurrences to the name.[21]

```haskell
-- | Makes sure that (block) names are unique.
uniqueName :: String -> Names -> (String, Names)
uniqueName nm ns = case Map.lookup nm ns of
    Nothing -> (nm,  Map.insert nm 1 ns)
    Just ix -> (nm ++ show ix, Map.insert nm (ix+1) ns)
```

The following functions deal with adding new blocks to the block stack, switching to a new active block to append incoming instructions to, and modifying existing blocks.

```haskell
emptyBlock :: Int -> BlockState
emptyBlock i = BlockState i [] Nothing

addBlock :: String -> Codegen Name
addBlock bname = do
    bls <- gets blocks
    ix <- gets blockCount
    nms <- gets names
    let new = emptyBlock ix
        (qname, supply) = uniqueName bname nms
    modify $ \s -> s { blocks = Map.insert (Name qname) new bls
                     , blockCount = ix + 1
                     , names = supply
                     }
    return (Name qname)

-- | Switches to a new active block to append to.
setBlock :: Name -> Codegen Name
setBlock bname = do
    modify $ \s -> s { currentBlock = bname }
    return bname

getBlock :: Codegen Name
getBlock = gets currentBlock

-- | Replaces the current block with a new block.
modifyBlock :: BlockState -> Codegen ()
modifyBlock new = do
    active <- getBlock
    modify $ \s -> s { blocks = Map.insert active new (blocks s) }

```

[21]This is required for nested `if/else` statements, for example. Looking at how they are implemented (see section 3.7.3), consider what would happen if the body of the `else` branch contained another `if/else` statement: it would contain duplicate block names without this deduplication.

```
32  -- | Sorts blocks based on their indices.
33  sortBlocks :: [(Name, BlockState)] → [(Name, BlockState)]
34  sortBlocks = sortBy (compare `on` (idx . snd))
```

All of this cannot be used easily without another state monad (this time wrapping a `CodegenState`) and related functions that help transform block states and codegen states to `llvm-general` basic blocks:

```
1   newtype Codegen a = Codegen
2       { runCodegen :: State CodegenState a
3       } deriving (Functor, Applicative, Monad, MonadState CodegenState)
4
5   -- | Name of the entry block of each function.
6   entryBlockName :: String
7   entryBlockName = "entry"
8
9   -- | Creates a block with the passed name and state. If the terminator
      ↪ is set to 'Nothing', throws an error.
10  makeBlock :: (Name, BlockState) → BasicBlock
11  makeBlock (l, BlockState _ s t) = BasicBlock l s (maketerm t)
12    where
13      maketerm (Just x) = x
14      maketerm Nothing = error $ "Block has no terminator: " ++ show l
15
16  -- | Given a codegen stage, sorts the contained blocks and transforms
      ↪ them to basic blocks.
17  createBlocks :: CodegenState → [BasicBlock]
18  createBlocks m = map makeBlock $ sortBlocks $ Map.toList (blocks m)
```

The backend of the framework is complemented with two more functions, the second of which essentially creates a `CodegenState` from an instance of the `Codegen` monad and a symbol table:

```
1   -- | Initial 'CodegenState'.
2   emptyCodegen :: CodegenState
3   emptyCodegen = CodegenState (Name entryBlockName) Map.empty [] 1 0
      ↪ Map.empty
4
5   -- | Evaluates an empty codegen state given the codegen 'm' and a symbol
      ↪ table.
6   execCodegen :: SymbolTable → Codegen a → CodegenState
7   execCodegen vars m = execState (runCodegen m) emptyCodegen { symtab =
      ↪ vars }
```

With a bunch of infrastructure for dealing with modules, definitions and blocks in place, we can finally devote some attention to placing instructions in them. The following functions aid in adding instructions to a block and assigning their return values to unnamed registers, as well as terminating blocks:[22]

```
1   -- | Gets the current block or throws an error if there is none.
2   current :: Codegen BlockState
3   current = do
4       c ← gets currentBlock
```

[22]LLVM blocks must be terminated with a special terminator instruction, see `http://llvm.org/docs/LangRef.html#terminator-instructions`.

```
5      blks ← gets blocks
6      case Map.lookup c blks of
7          Just x → return x
8          Nothing → error $ "No such block: " ++ show c
9
10 -- | Creates new internal variable name/register number.
11 fresh :: Codegen Word
12 fresh = do
13     i ← gets count
14     modify $ \s → s { count = 1 + i }
15     return $ i + 1
16
17 -- | Adds an instruction to the current block's stack.
18 instr :: Instruction → Codegen Operand
19 instr ins = do
20     n ← fresh
21     let ref = UnName n
22     blk ← current
23     let i = stack blk
24     modifyBlock (blk { stack = i ++ [ref := ins] } )
25     return $ local ref
26
27 -- | Sets the terminator of the current block.
28 terminator :: Named Terminator → Codegen (Named Terminator)
29 terminator trm = do
30     blk ← current
31     modifyBlock (blk { term = Just trm })
32     return trm
```

In order to complete the core of the code generation framework, we need to be able to efficiently modify the symbol table contained in every `CodegenState`:

```
1  -- | Assigns an operand (i.e. its address) to a local variable name.
2  assign :: Varname → Operand → Codegen ()
3  assign var x = do
4      lcls ← gets symtab
5      modify $ \s → s { symtab = (var, (Nothing, x)) : lcls }
6
7  -- | Infix operator for 'assign'. The @*@ signifies that the second
    ↪  argument should be a pointer.
8  (<:*) :: Varname → Operand → Codegen ()
9  var <:* ptr = assign var ptr
10
11 -- | Assigns an operand (i.e. its address) with an associated KL type to
    ↪  a local variable name.
12 assignAs :: K.TyValue → Varname → Operand → Codegen ()
13 assignAs ty var x = do
14     lcls ← gets symtab
15     modify $ \s → s { symtab = (var, (Just ty, x)) : lcls }
16
17 -- | Infix operator for 'assignAs'. The @#@ signifies that 'ty' is a
    ↪  'K.TyValue' while @*@ indicates that 'ptr' should be a pointer.
18 (<:#*) :: Varname → (K.TyValue, Operand) → Codegen ()
19 var <:#* (ty, ptr) = assignAs ty var ptr
20
21 -- | Returns the associated operand or throws an error if the variable
    ↪  is not set in the current scope.
22 getvar :: Varname → Codegen Operand
23 getvar var = do
```

```
    syms ← gets symtab
    case lookup var syms of
        Just (_, x) → return x
        Nothing     → error $ "Local variable not in scope: " ++ show
          ↪  var

-- | Returns the KL type of the variable or throws an error of the
  ↪  variable was not given a type or does not exist.
gettype :: Varname → Codegen K.TyValue
gettype var = do
    syms ← gets symtab
    case lookup var syms of
        Just (Just ty, _) → return ty
        Just (Nothing, _) → error $ "Local variable was not assigned a
          ↪  type: " ++ show var
        Nothing           → error $ "Local variable not in scope: " ++
          ↪  show var
```

As well as references to local, global and external definitions:

```haskell
-- | Typed local variable/reference.
localTy :: Type → Name → Operand
localTy = LocalReference

-- | Typed global variable/reference.
globalTy :: Type → Name → C.Constant
globalTy = C.GlobalReference

-- | Typed external function/reference.
externfTy :: Type → Name → Operand
externfTy ty = ConstantOperand . C.GlobalReference ty
```

3.5.2 Utilities

Some of the less essential code from the tutorial, as well as more of my own code, can be found in the `Codegen.Util` module. For brevity's sake, I will only highlight the most important bits here.

The `ToConstant` type class I added to the framework is related to the local and global references discussed at the end of the previous section. It provides two functions that convert the class instances to LLVM constants and operands, respectively.

```haskell
class ToConstant a where

    -- | Conversion to an LLVM IR constant.
    toConstant :: a → C.Constant

    -- | Conversion to an LLVM IR operand.
    toOperand :: a → Operand
    toOperand = ConstantOperand . toConstant

instance ToConstant Bool where
    toConstant True  = C.Int 1 1
    toConstant False = C.Int 1 0

instance ToConstant Int where
    toConstant i  = C.Int (fromIntegral intBits) (fromIntegral i)
```

In the previous listing, two instances are shown as an example. In the ToConstant instance for integers, the intBits function reads the size of an integer on the current platform from a config file (see section 3.10).

The following functions help with allocating heap memory[23] and wrting/reading values to/from memory. Don't confuse the store function with the assign function discussed earlier: the latter binds a pointer to a name, while the former stores data in such a pointer.

```haskell
alloc :: Type → Codegen Operand
alloc ty = do
    let numBytes = cons $ (C.Int $ fromIntegral sizetBits) (fromIntegral
      $ sizeof ty)
    a ← callCbits voidPtr "kltollvm_alloc" [numBytes]
    bitcast (T.ptr ty) a

-- | Store a value in the memory pointed to.
store :: Operand → Operand → Codegen Operand
store ptr val = instr $ Store False ptr val Nothing 0 []

-- | Infix operator for 'store'. The @*@ signifies that the first
  operand should be a pointer.
(*<~) :: Operand → Operand → Codegen Operand
ptr *<~ val = store ptr val

-- | Load from a pointer.
load :: Operand → Codegen Operand
load ptr = instr $ Load False ptr Nothing 0 []
```

The following two functions are helpful for passing data to the polymorphous functions throughout the standard library: toVoidPtr makes sure that the input operand is "wrapped", i.e. pointed to by a void pointer. For non-atomic types, this involves storing them on the heap. Anything else should already be in the form of a pointer. The fromVoidPtr function acts as the inverse, "unwrapping" atomic values.

```haskell
toVoidPtr :: K.TyValue → Operand → Codegen Operand
toVoidPtr ty a = case ty of
    K.TyBool  → callCbits voidPtr "kltollvm_from_bool" [a]
    K.TyInt   → callCbits voidPtr "kltollvm_from_int" [a]
    K.TyFloat → callCbits voidPtr "kltollvm_from_float" [a]
    _         → return a

fromVoidPtr :: K.TyValue → Operand → Codegen Operand
fromVoidPtr ty a = case ty of
    K.TyBool  → bitcast (T.ptr Ty.bool) a >>= load
    K.TyInt   → bitcast (T.ptr Ty.int) a >>= load
    K.TyFloat → bitcast (T.ptr Ty.float) a >>= load
    _         → return a
```

Being able to compile Haskell lists to LLVM arrays is necessary when dealing with strings, but it also comes in handy during code generation for row literals. My implementation uses an internal variable to store the current position in the array (which is incremented using the gep function) during the insertion loop.

[23] Notice the callCbits function used here – it calls the function in its second argument with the argument list from its third argument.

```haskell
-- | Given a list of operands @l@, all of which must be of type @ty@,
  ↪ return a pointer @'T.ptr' ty@ to an array containing all operands.
listToArray :: T.Type → [Operand] → Codegen Operand
listToArray ty l = do
    let ivar = "kltollvm_listtoarray_internal"

    -- Allocate space for array with correct length, taking the type
      ↪ into account.
    mem ← alloc $ arrayOf (length l) ty

    -- Get pointer to first element.
    i ← gep mem [toOperand (0 :: Int), toOperand (0 :: Int)]
    ivar <:* i

    -- For each element of the list, we store it at the current position
      ↪ in the array and advance the pointer by 1.
    forM_ l $ \lx → do
        i ← getvar ivar
        i *<~ lx
        i ← gep i [toOperand (1 :: Int)]
        ivar <:* i

    bitcast (T.ptr ty) mem

-- | Code generation for string literals. A null byte/terminator is
  ↪ added automatically.
cgenStr :: String → Codegen Operand
cgenStr s = do
    let chars = map toOperand $ s ++ "\0"
    arr ← listToArray char chars
    bitcast string arr
```

This module also contains the logging helpers introduced in section 3.3.

3.5.3 Arithmetic, Comparison and Typecasts

The `Codegen.Op` module contains some convenience functions wrapping LLVM instructions related to basic builtins operators. Included are arithmetic and comparison for both integers and floating-point numbers, casts between floating point and integer types, as well as the standard boolean operators.

As an example, the boolean `not` operator is implemented as follows (see the highlighted line):

```haskell
true :: Operand
true = toOperand True

-- | Not part of KL, but useful for defining NOT.
bxor :: Operand → Operand → Codegen Operand
bxor a b = instr $ Xor a b []

bnot :: [Operand] → Codegen Operand
bnot [a] = bxor a true
```

3.6 Functions

Code generation for functions[24] looks fairly complex at first. Below the code listing, I will explain what each line does.

```
-- | Top-level code generation for a typed KL function.
cgenDef :: Logging -> K.TyStmt -> LLVM ()
cgenDef l (K.Def name args body retexpr (Just (K.TyFun argtys retty))) =
  ↪ define lretty name largs bls
  where
    lretty = tyValueToLlvm retty
    largs = toSig argtys args
    bls = createBlocks $ execCodegen [] $ do
        entry <- addBlock entryBlockName
        setBlock entry
        forM_ (zip argtys args) $ \(ty,a) -> do
            var <- alloc $ tyValueToLlvm ty

            -- Copy complex values when passed to functions to simulate
            -- pass-by-value.
            val <- if (not . isAtomic) ty
                then deepcopy ty $ local $ AST.Name a
                else return $ local $ AST.Name a

            var *<~ val
            a <:#* (ty, var)
        a <- if l && name == "main"
            then do
                initCbits
                initLogging
                cgen l body
                finishLogging
            else cgen l body
        d <- cgenExpr l retexpr
        ret d
```

Line 3 uses pattern-matching to extract the required values from the AST and calls the `define` function discussed in section 3.5.1 to assemble a LLVM function definition. In line 5, the KL return type is translated to its LLVM equivalent, and in the following line, the function signature is converted to the desired format. Lines 8 and 9 create the function's entry block and set it to be the active block, meaning that and subsequently created instructions are appended to it. In the code generated by the `forM_` "loop" starting in line 10, space for each function argument is allocated, non-atomic arguments are copied (atomic arguments follow the desired pass-by-value semantics without this extra work), the arguments are stored in the allocated space, and finally, a pointer to the memory now containing the argument are assigned to the variable name associated with the argument.

Before considering lines 22 through 28, note that all top-level (naked) statements in the KL AST are wrapped in a `main` function as a preprocessing step. If the function to be compiled here is this `main` function, the garbage collector and log files are initialized before the function body is compiled, and the log files are finished up afterwards. For all other functions, only the body is compiled (see line 28).

Finally, the return expression `retexpr` is compiled (given the environment built up by the previous steps) and set as the return value of the function.

[24] Located in `KLToLLVM/code/src/Codegen/Codegen.hs`.

3.7 Statements

The `cgen` function in the `Codegen` module is responsible for handling statements. Its implementation employs pattern matches to provide different definitions for each KL statement.

3.7.1 Variable Declaration

Code generation for variable declarations in KL (e.g. `var a : int;`) is very simple, only allocating the required space (without storing anything yet) and assigning it to the desired variable name:

```haskell
cgen :: Logging -> [K.TyStmt] -> Codegen AST.Operand
cgen l (K.Var var ty : xs) = do
    i <- alloc $ tyValueToLlvm ty
    var <:#* (ty, i)
    cgen l xs
```

3.7.2 Variable Assignment

Actually assigning values to variables after they have been declared is somewhat more involved, largely due to ensuring that non-atomic values are copied when read directly from another variable, as well as a small memory optimization:[25]

```haskell
cgen l (K.Put var b : xs) = do
    let ty = K.typeOf b
    unless (ty == K.TyNull) $ do  -- If the value being assigned is null, do nothing.
        val <- case b of
            K.TGet v _ -> do  -- Deepcopy on a = b assignment where a, b are non-atomic variables.
                val' <- getvar v >>= load
                if (not . isAtomic) ty
                    then deepcopy ty val'
                    else return val'
            _ -> cgenExpr l b
        i <- getvar var
        unless (isAtomic ty) $ do  -- Delete old value.
            j <- load i
            callCbits T.void "kltollvm_dealloc" [j]
        i *<~ val
        var <:#* (ty, i)
        noop
    cgen l xs
```

3.7.3 Conditional

The code generator for the venerable `if/else` statement showcases creating and switching between LLVM blocks neatly:

[25] The noop in the penultimate line is required for the do block to return a `Codegen Operand` instead of `Codegen ()`.

```haskell
cgen l (K.If cond tr fl : xs) = do
    ifthen <- addBlock "if.then"
    ifelse <- addBlock "if.else"
    ifexit <- addBlock "if.exit"

    -- entry
    test <- cgenExpr l cond
    lcput l test
    cbr test ifthen ifelse

    -- if.then
    setBlock ifthen
    cgen l tr
    br ifexit
    ifthen <- getBlock

    -- if.else
    setBlock ifelse
    cgen l fl
    br ifexit
    ifelse <- getBlock

    -- if.exit
    setBlock ifexit
    cgen l xs
```

First, the three required blocks (which will later serve as jump targets) are created. Then, the boolean expression `cond` is compiled and the associated operand is stored in `test` before it is written to the control flow log. Then a conditional jump instruction is generated, which will redirect the control flow to the `ifthen` block if `cond` evaluates to true, and otherwise to `ifelse`. In each block, the corresponding KL statements are compiled before an unconditional jump to the `ifexit` block.

The reason for calling the `getBlock` function at the end of each block "is that the [`if/then` statement] may actually itself change the block that the Builder is emitting into if, for example, it contains a nested [`if/then` statement]. Because calling `cgen` recursively could arbitrarily change the notion of the current block, we are required to get an up-to-date value". [Die]

3.7.4 Loops

There are two kinds of loops in KL: a `while` loop that repeatedly evaluates a boolean expression and a `for` loop that iterates through a list, assigning the current element to a loop variable to be used in the loop body. Neither of the loops supports early termination (i.e. a break statement), which makes the implementation relatively straightforward.

For this reason, I'll skip the `while` loop (it's very close to `if/else`) and go straight to the more complex `for` loop. Its implementation lives in a separate function because much of the same functionality is required for logging the return value of the `keys` builtin.[26] First, consider the type of the `forLoop` function and the blocks required by my approach:

[26] This is required because dictionary ordering is implementation-dependant – as a result, the list of keys could be in an arbitrary order. In order to enable provenance derivation, the order needs to be known, however, so the list of keys needs to be traversed and each element recorded in the index log. The cleanest way to achieve this (with minimal code duplication) was to introduce a special case into the `for` loop implementation.

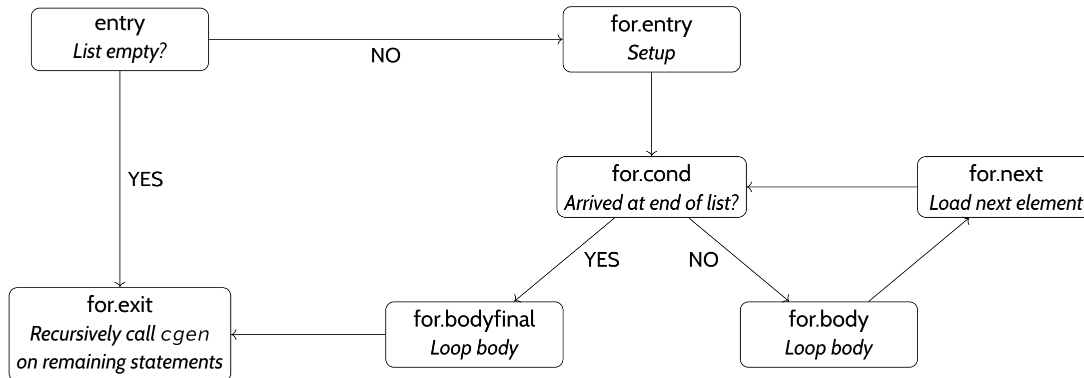

Figure 3.1: Flowchart-like representation of the blocks and the control flow between them.

```haskell
forLoop :: Logging
        → String                  -- ^ Loop variable name
        → K.TyValue               -- ^ Type of list elements
        → AST.Operand             -- ^ The list to iterate over
        → Maybe [K.TyStmt]        -- ^ Loop body
        → Bool                    -- ^ Whether to log the loop variable in
                                  -- each iteration
        → Codegen AST.Operand
forLoop l varname elemty li body logElements = do
    forentry ← addBlock "for.entry"
    forcond ← addBlock "for.cond"
    forbody ← addBlock "for.body"
    forbodyfinal ← addBlock "for.bodyfinal"
    fornext ← addBlock "for.next"
    forexit ← addBlock "for.exit"
```

The body is of type `Maybe [K.TyStmt]` because of the aforementioned logging of the `keys` builtin's return value – when performing this task, there is no body to generate code from. For the same reason, a boolean switch `logElements` is provided. This will become relevant later when going over the implementation of the `forbody` and `forbodyfinal` blocks.[27]

To make the implementation easier to understand, figure 3.1 shows what each of the blocks in the listing above do and how they are connected in terms of control flow.

Continuing from the previous code snippet, I will let the heavily commented code speak for itself for a bit:

```haskell
    -- Check if the list is empty. If it is, jump straight to the exit.
    len ← callCbits int "kltollvm_list_for_length" [li]
    test ← ieq [len, toOperand (0 :: Int)]
    cbr test forexit forentry

    -- for.entry
    setBlock forentry

    -- Allocate memory for the loop variable.
    mem ← alloc $ tyValueToLlvm elemty
    varname <:#* (elemty, mem)

```

[27] These two blocks could be unified into one – see section 5.5.

```
13      -- We loop through the list with the help of the C functions
        ↪ "kltollvm_list_for_first" (returning a pointer to the internal
        ↪ representation of the first list element),
        ↪ "kltollm_list_for_next" (which given a pointer to the internal
        ↪ representation of a list element, returns a pointer to the next
        ↪ one) and "kltollm_list_for_last" (see "..._first"). So we need
        ↪ to save the current list element i in the following variable:
14      let internalvarname = "kltollvm_for_internal_" ++ varname
15      mem <- alloc voidPtr
16      internalvarname <:* mem
17      i <- callCbits voidPtr "kltollvm_list_for_first" [li]
18      internalforvar <- getvar internalvarname
19      internalforvar *<~ i
20      j <- callCbits voidPtr "kltollvm_list_for_last" [li]
21      br forcond
22
23      -- for.cond
24      setBlock forcond
25      i <- getvar internalvarname >>= load
26
27      -- Get the value of the curent list element i.
28      forval' <- callCbits voidPtr "kltollvm_list_for_value" [i]
29      forval <- fromVoidPtr elemty forval'
30      forvar <- getvar varname
31      forvar *<~ forval  -- Super important! Need to store before
        ↪ assigning.
32      varname <:#* (elemty, forvar)
33
34      -- Compare pointer to current list element with pointer to last
        ↪ element, and jump to the final iteration if they are the same.
35      test <- ieq [i, j]
36      cbr test forbodyfinal forbody
```

The previous two lines are the reason for the very similar (see below) `forbody` and `forbodyfinal` blocks: upon completion, `forbody` jumps to `fornext`, while `forbodyfinal` jumps to the exit. Apart from this, they generate the same LLVM IR code, first compiling the body of the loop if it exists, and then, if the `logElements` flag is enabled, writing the current list element to the index log:

```
1       -- for.body
2       setBlock forbody
3       case body of
4           Just stmts -> cgen l stmts
5           Nothing -> noop
6       when logElements $ do
7           key <- getvar varname >>= load
8           liput l key $ Left elemty
9       br fornext
10      forbody <- getBlock
11
12      -- for.bodyfinal
13      setBlock forbodyfinal
14      case body of
15          Just stmts -> cgen l stmts
16          Nothing -> noop
17      when logElements $ do
18          key <- getvar varname >>= load
19          liput l key $ Left elemty
20      br forexit
```

```
21      forbodyfinal ← getBlock
22
23      -- for.next
24      setBlock fornext
25
26      -- Load the next list element.
27      i ← callCbits voidPtr "kltollvm_list_for_next" [li, i]
28      internalforvar *<~ i
29      br forcond
30
31      -- for.exit
32      setBlock forexit
33      noop
```

Finally, returning to the cgen function, we can see how the forLoop function discussed above is called:

```
1  cgen l (K.For varname list body : xs) = do
2      let elemty = containedType $ K.typeOf list
3      li ← cgenExpr l list
4      forLoop l varname elemty li (Just body) False
5      cgen l xs
```

3.7.5 Append and Update

Appending a new value to a list is straightforward:

```
1  cgen l (K.Append varlist expr : xs) = do
2      li ← getvar varlist >>= load
3      value ← cgenExpr l expr >>= toVoidPtr (K.typeOf expr)
4      callCbits T.void "kltollvm_list_append" [li, value]
5      cgen l xs
```

First, the pointer to the list stored in the symbol table is dereferenced. Then, the value to be inserted is generated by evaluating the corresponding expression, making sure it is "wrapped" in a pointer using the toVoidPtr function discussed in section 3.5.2. Then the relevant C function is called before, as usual, the rest of the statements are processed.

Updates are slightly more complex primarily because a key is involved in addition to a new value, as well as the need to discern between dicts and rows. First, the dict (or row) is loaded. Then, the key expression is evaluated and written to the index log file, before again being pointer-"wrapped". The new value receives the same treatment modulo logging. Before the case statement, we can see why saving the type of each variable in the symbol table (see section 3.5.1) was worth it: otherwise, there would be no way to tell whether to perform a dict or row update.

```
1  cgen l (K.Update vardictrec keyexpr newvalexpr : xs) = do
2      dr ← getvar vardictrec >>= load
3      key' ← cgenExpr l keyexpr
4      liput l key' $ Right keyexpr
5      key ← toVoidPtr (K.typeOf keyexpr) key'
6      value ← cgenExpr l newvalexpr >>= toVoidPtr (K.typeOf newvalexpr)
7
8      drty ← gettype vardictrec
```

```
9       case drty of
10          K.TyDict{} → callCbits T.void "kltollvm_dict_update" [dr, key,
             ↪ value]
11          K.TyRow{} → do
12              let keytype = (toOperand . tyValueToC . K.typeOf) keyexpr
13              callCbits T.void "kltollvm_row_update" [dr, keytype, key,
                 ↪ value]
14
15      cgen l xs
```

3.7.6 Printing and Assertions

Because print statements and assertions are only helpful during development and for debugging purposes,[28] I will skip printing, instead briefly explaining how assertions are implemented:

```
1   cgen l (K.Assert cond : xs) = do
2       assertpassed ← addBlock "assert.passed"
3       assertfailed ← addBlock "assert.failed"
4
5       -- Evaluate assertion. Note that logging is disabled here.
6       test ← cgenExpr False cond
7       cbr test assertpassed assertfailed
8
9       -- assert.failed
10      setBlock assertfailed
11
12      -- Pretty-print cond into string, print it and exit.
13      let s = "Assertion failed: " ++ (show . pretty) cond
14      exitWithMessage s
15      unreachable
16      assertfailed ← getBlock
17
18      -- assert.passed
19      setBlock assertpassed
20      cgen l xs
```

Similar to `if/else` statements, assertions can lead down two paths: if the assertion passes, all is well and the program merrily continues along its way. If it doesn't, the failed assertion is printed, which involves calling the research group-internal `kl-prettyprinter` package into action in line 13[29] before execution is terminated. The "nullary terminator instruction"[30] inserted by calling the `unreachable` function is helpful here because LLVM blocks need to be capped off with special terminator instructions.[31]

3.8 Expressions

Expression code generation is handled by the `cgenExpr` function.

[28]Neither directly impacts data provenance, which is reflected in logging being disabled in the arguments of theses statements.

[29]Simply printing the offending AST node would be possible, but harder to parse for the user trying to fix whatever led to the failed assertion.

[30]Discussed by Chris Lattner in one of his "Random LLVM Notes", see `http://nondot.org/sabre/LLVMNotes/UnreachableInstruction.txt`.

[31]Documented at `http://llvm.org/docs/LangRef.html#terminator-instructions`.

3.8.1 Literals

For atomic literals, the `toOperand` function is called. String literals, which are used in other places in the codebase as well, are compiled by the `cgenStr` function described in section 3.5.2.

```
cgenExpr l (K.TBool b ty) = return $ toOperand b
cgenExpr l (K.TInt i ty) = return $ toOperand i
cgenExpr l (K.TFloat f ty) = return $ toOperand f
cgenExpr l (K.TStr s ty) = cgenStr s
```

When the KL to LLVM compiler encounters an untyped list or dict literal, an error is thrown:[32]

```
cgenExpr l (K.TList li (K.TyList K.TyNull)) = error "Found an untyped
  ↪ empty list literal ([] without a type annotation), please change it
  ↪ to [] : [a] where a is the desired element type"

cgenExpr l (K.TDict d (K.TyDict K.TyNull K.TyNull)) = error "Found an
  ↪ untyped empty dict literal ({} without a type annotation), please
  ↪ change it to {} : {a : b} where a and b are the desired key and
  ↪ value types, respectively"
```

For brevity's sake, I will skip how list and row literals are compiled (because it's fairly simple) and focus on dict literals:

```
cgenExpr l (K.TDict d ty) = do

    -- Create an empty dictionary.
    let kty = tyValueToC $ keyType ty
    let vty = tyValueToC $ valueType ty
    a <- callCbits voidPtr "kltollvm_dict_empty" $ map toOperand [kty,
      ↪ vty]

    -- Insert key-value pairs.
    forM_ d $ \(dk, dv) -> do
        dxk <- cgenExpr l dk
        liput l dxk $ Right dk
        dxv <- cgenExpr l dv

        vdxk <- toVoidPtr (K.typeOf dk) dxk
        vdxv <- toVoidPtr (K.typeOf dv) dxv

        callCbits T.void "kltollvm_dict_update" [a, vdxk, vdxv]

    return a
  where
    keyType   (K.TyDict a b) = a
    valueType (K.TyDict a b) = b
```

First, the key and value types are translated to the constants used throughout the standard library (see section 3.4), before an empty dictionary is created based on these types. Then, for each key-value pair, the key is logged and both key and value are

[32]This is because the type checker does not provide a type for empty literals, and the KL to LLVM compiler needs to know which type to pass to the `kltollvm_list_empty()` function. This problem was solved by adding support for type annotations for empty literals to the KL parser.

compiled before the `kltollvm_dict_update()` function is called to insert them into the dictionary.

3.8.2 List Operations

KL defines the following three expressions acting on lists:

```
1  cgenExpr l (K.TElemL e1 e2 ty) = do
2      a ← cgenExpr l e1
3      b ← cgenExpr l e2
4      liput l b $ Right e2
5      c ← callCbits voidPtr "kltollvm_list_eleml" [a, b]
6      fromVoidPtr ty c
7
8  cgenExpr l (K.TLength e ty) = do
9      a ← cgenExpr l e
10     callCbits int "kltollvm_list_length" [a]
11
12 cgenExpr l (K.TContains e1 e2 ty) = do
13     a1 ← cgenExpr l e1
14     a2 ← cgenExpr l e2
15     a2' ← toVoidPtr (K.typeOf e2) a2
16     callCbits bool "kltollvm_list_contains" [a1, a2']
```

All of them basically boil down to compiling the inputs and calling the corresponding C function. The first one corresponds to element access like `list[elem]`, hence the `liput` call – also note that the `fromVoidPointer` call in line 6 dereferences pointers to atomic values.

3.8.3 Dict/Row Operations

Apart from dict/row access expressions which are implemented analogously to list element access, the `keys()` builtin returns a list containing the key set of a dictionary, which are written to the index log using a `forLoop` as described in section 3.7.4:[33]

```
1  cgenExpr l (K.TKeys e ty) = do
2      a ← cgenExpr l e
3      li ← callCbits voidPtr "kltollvm_dict_keys" [a]
4      when l $ do
5          let varname = "kltollvm_keys_key"
6          let elemty = containedType ty
7          forLoop l varname elemty li Nothing True
8      return li
```

3.8.4 Miscellaneous

The two remaining KL expressions to be discussed are variable access and function calls. Variables are accessed by loading the associated memory address as a pointer and derefencing it. Code generation for function calls consists of first compiling the arguments and then generating an LLVM function call instruction with the correct return type and the compiled arguments.

[33] This builtin is not defined on rows, whose keys can be of inhomogenous types. A KL list, meanwhile, can only contain values of a single type.

```haskell
cgenExpr l (K.TGet v ty) = getvar v >>= load

cgenExpr l (K.TCall f es ty) = do
    largs <- mapM (cgenExpr l) es
    call (externfTy (tyValueToLlvm ty) (AST.Name f)) largs
```

3.9 Operators

Code generation for built-in operators is taken care of by the `cgenBuiltinOp` function. Given an operator and its list of arguments, it calls the `boLookup` function which checks if the operator is valid and whether the length of the argument list matches the operator's arity.

```haskell
cgenBuiltinOp :: Logging -> K.TyExpr -> Codegen AST.Operand
cgenBuiltinOp l o@(K.TOper op opands ty) = case boLookup op of
    Nothing -> error $ "Operator not in builtinOperators: " ++ show o
    Just bo -> do
        operands <- mapM (cgenExpr l) opands
        let opTy = K.typeOf $ head opands
        when (length opands /= fromIntegral (K.opArity bo)) $ error $
          ↪  "Arity is " ++ show (length opands) ++ "should be " ++ show
          ↪  (K.opArity bo) ++ " for the operator: " ++ show o
        case op of
            ...
```

Then, in most cases, a function (either wrapping an LLVM instruction or a C function) implementing the relevant operator is called. Operators that handle numbers mostly have different implementations depending on whether the return value is an integer or a float. For example, consider the implementation of the $\leq$ operator below. Booleans are represented as 1-bit integers, so for both types the `ilte` function is used to emit an appropriate `icmp` instruction.[34] Floating-point comparison functions similarly, while strings are compared using a C function.

```haskell
            ...
            K.Lte -> case opTy of
                K.TyBool   -> ilte operands
                K.TyInt    -> ilte operands
                K.TyFloat  -> flte operands
                K.TyStr    -> callCbits bool "kltollvm_str_lte" operands
            ...
```

3.10 Miscellaneous

In this section, I will give a brief overview over the not-yet-mentioned modules, followed by a brief dive into the configuration of the compiler via its config file and command-line options. Finally, I will go through the internal compilation pipeline as implemented in the `Main` module.

- The `Preprocess` module contains a few small preprocessing passes, mainly grouping top-level expressions into a `main()` function.

[34] Documentation available at `http://llvm.org/docs/LangRef.html#icmp-instruction`.

- The Emit module manages loading the standard library from a file (as mentioned in section 3.1.6) into an llvm-general AST, kicking off the code generation process on top of this AST (see below) and serializing the final AST comprised of the standard library and compiled KL code back to the textual representation of LLVM IR.

```
compile :: Logging → [K.TyStmt] → IO AST.Module
compile l kl = do
    stdl ← loadStdlib
    let ast = codegen l stdl kl
    return ast
```

- Output helper functions are provided by the Messages module. Some of them depend on the verbosity level...

- ...which can be set using command-line flags. The command-line interface (CLI) of the KL to LLVM compiler is implemented in the CLI module using the popular optparse-applicative package, allowing the user to set the optimization level, enable logging, and choose the output format. Running kltollvm -h prints a summary of the available options.

- Supplementing the CLI, the Config module provides access to some options that the user is likely to change less frequently. Located at KLToLLVM/code/config.ini, the configuration file contains paths to required directories and executables, the linker flags required for using GLib and the Boehm GC, as well as the sizes of fundamental data types as outlined in section 3.4.[35]

- The Main module ties everything together, running the CLI parser, reading and verifying the config file and then performing the necessary compilation steps in sequence. It relies on three shell scripts located in KLToLLVM/code/scripts/ to build the standard library (see section 3.1.6), run LLVM's standard optimization passes on the generated LLVM IR code, and create an executable file (if the corresponding flag is given).

[35] A heavily commented template is provided at KLToLLVM/code/config-blank.ini.

"There is no programming language, no matter how structured, that will prevent programmers from making bad programs."

— Larry Flon[1]

Chapter 4

Discussion

In this chapter, I will first dive into how the KL to LLVM compiler performs compared to the interpreter, taking a look at both compilation times and runtimes of emitted executables. In addition, I will consider how the different data structure implementations stack up against each other.

Next, I will take a critical look at some aspects of the code I've written in the context of this thesis, and how efficient the generated LLVM IR code is. Concluding this chapter, I will talk about some of the challenges I faced during the last four+ months.

4.1 Performance & Benchmarks

Before testing how the KL to LLVM compiler stacks up against the KL interpreter, I will compare how the different list and dictionary implementations compare against each other. In order to find out, I wrote a simple shell script that, given a KL file, loops through different list lengths or dict sizes and successively inserts them into a specific line of the KL code. Filling the list or dict with pseudo-random values without adding significant overhead proved to be difficult, so I temporarily changed code generation for a builtin operator to call a C function returning a random number. The shell script then uses this minimally modified KL to LLVM compiler to compile the KL file and subsequently runs the resulting binary a set number of times to cancel out any measuring errors.

The KL file used for benchmarking lists is an implementation of Quicksort,[2] which I have tested for lists of varying length, starting with 1000 and going up to 20000 in increments of 1000. These tests have been re-run without logging to show the overhead it adds, which is especially significant for the faster arraylist implementation: it more than doubles the time taken by the algorithm as shown in figure 4.1.

Comparing the dotted and solid red lines in the figure, the garbage collector actually slightly increases performance. This was noticed by its developers as well: "Performance of the nonincremental collector is typically competitive with malloc/free implementations. Both space and time overhead are likely to be only slightly higher for

[1] From `http://alvinalexander.com/text/there-no-programming-language-will-prevent-programmers-making-bad-programs`.

[2] It can be found at `KLToLLVM/thesis/benchmarks/qsort.kl`.

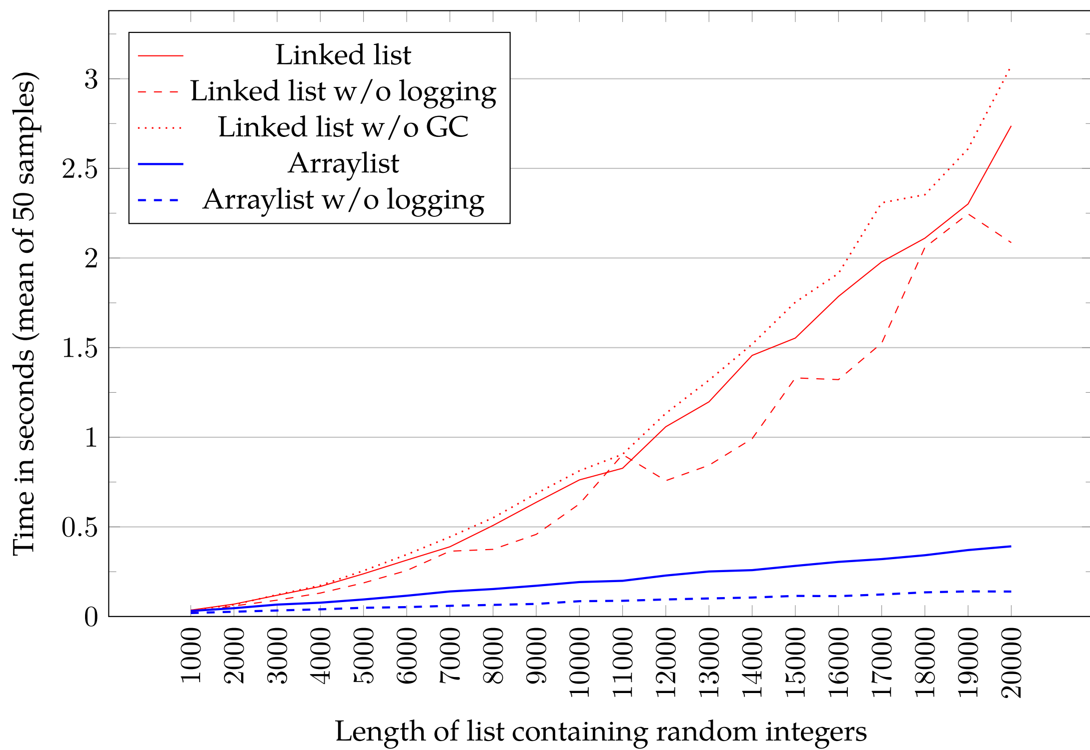

Figure 4.1: Runtimes of Quicksort, comparing different list implementations and how disabling other compiler features impacts performance.

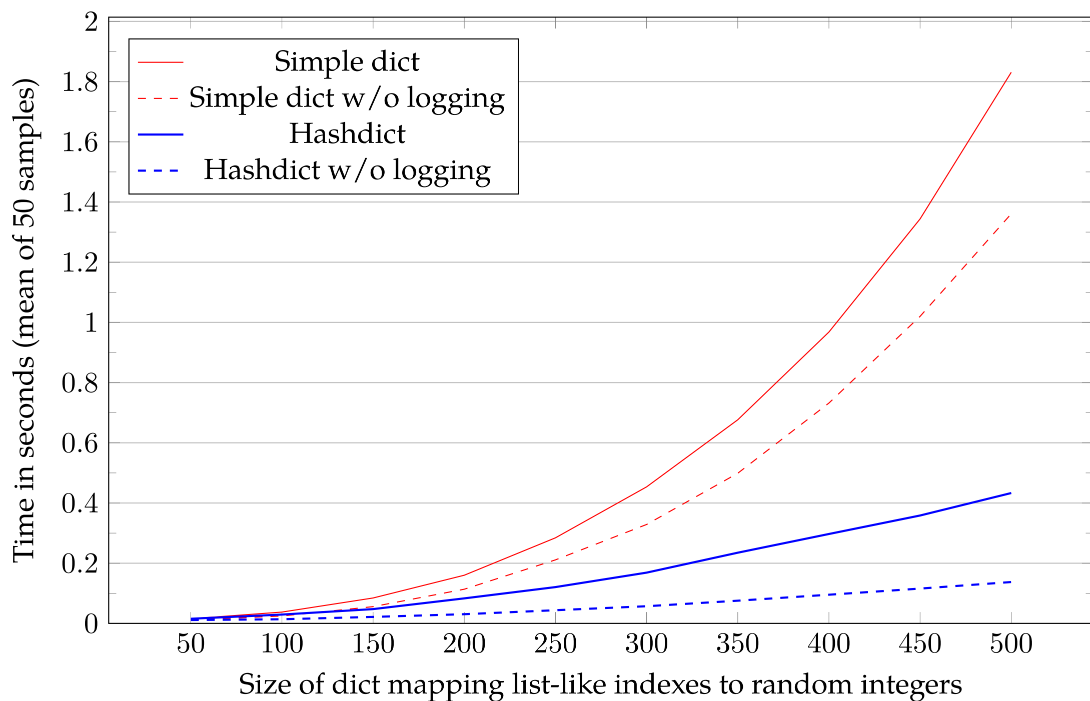

Figure 4.2: Runtimes of Bubblesort, comparing different dict implementations.

programs written for malloc/free [...]. For programs allocating primarily very small objects, the collector may be faster; for programs allocating primarily large objects it will be slower."[3]

For testing the performace of the two dict implementations detailed in sections 3.1.4 and 3.1.5, I implemented an in-place version of the Bubblesort algorithm in KL.[4] The results of my tests are shown in figure 4.2, clearly indicating that the hash-based dict implementation is more efficient by a large margin, especially when handling large dicts. Because Bubblesort is significantly slower than Quicksort, only dicts containing up to 500 elements were tested. Also note that the GLib-based dict implementation currently does not support garbage collection (see section 3.2), so the Boehm GC was disabled for all four tests.

Because the sorting algorithm implementations examined above do not represent real-world workloads very well, I have used different KL files for comparing my compiler to the interpreter. All four are automated translations of advanced SQL queries[5] designed to explore the derivation of helpful data provenance.

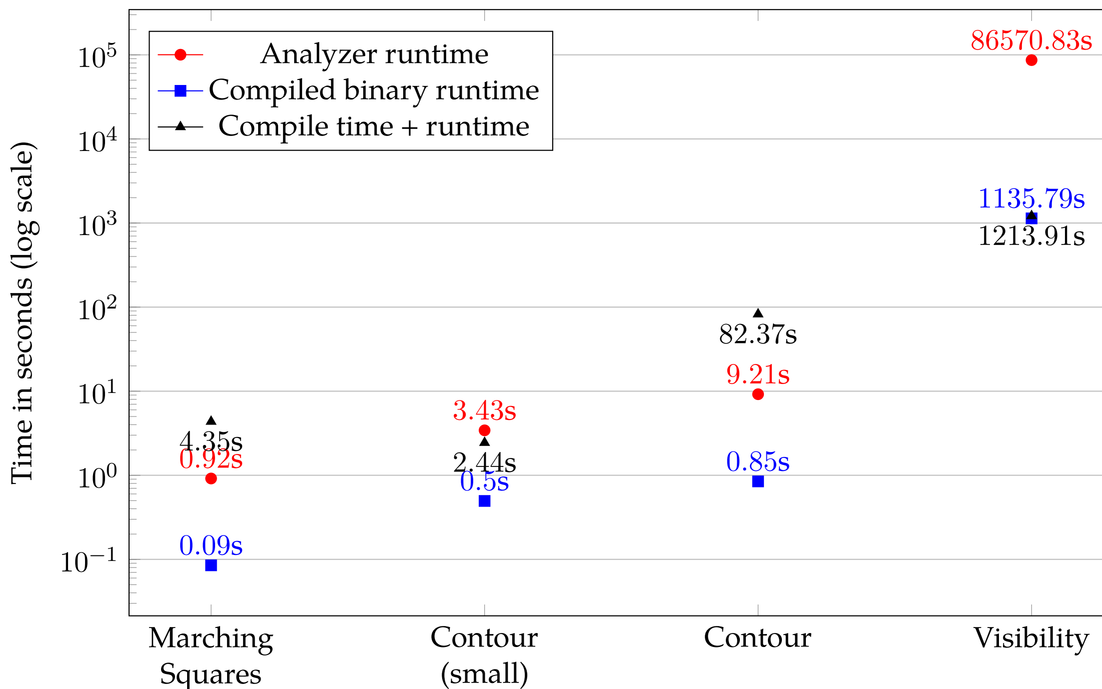

Figure 4.3: Comparison of compiler and interpreter based on real-world workloads.

Figure 4.3 shows how much time the execution of these queries took in my tests,[6] painting a mixed picture. (Note that the plot uses a logarithmic vertical axis to readably fit all measurements into a single figure.) While the binaries emitted by the KL to LLVM compiler are roughly an order of magnitude faster than the interpreter, when taking the compile time into account as well, this advantage melts away; the interpreter ends up

[3]Quote taken from `http://www.hboehm.info/gc/#details`.

[4]It can be found at `KLToLLVM/thesis/benchmarks/bubblesort-dict.kl`.

[5]The queries have been posted by members of the research group in a Slack channel and have been translated to KL by interim versions of Denis Hirn's SQL to KL compiler. The KL files can be found in the `KLToLLVM/code/examples` directory.

[6]With logging enabled in both the KL to LLVM compiler and analyzer. Additionally, the compiler was configured to use the faster list implemetation and the Boehm GC, forcing the use of the non-GLib dict and row implementations.

taking the lead for two queries.

I have found that compilation times seem to largely depend on the number of literals present: The rightmost two queries in the figure, which took longer than a minute to compile each, operate on 20x20 grids which are given as a list of rows containing three key-value pairs, e.g. `<|"x":20, "y":12, "alt":200|>`.[7] The smaller version of the Contour query compiles much faster, with the only change being a smaller 9x5 grid. This indicates that the compiler is currently most suited for computation-heavy queries that operate on small input data sets.

Performance of the Visibility query[8] is particularly noteworthy. Even though it takes a little over a minute to compile and then almost another 19 minutes to run the emitted binary, it blows the interpreter, which takes more than a day in my testing, out of the water, beating it by almost two orders of magnitude. Note that peak memory usage of the executable was about 6 GB while the interpreter hovered around 4.5 GB.

See section 5.5 for further discussion of compiler performance.

4.2 Code Quality

In terms of coding style, I have roughly followed the style guide published by Johan Tibell,[9] however I chose not to strictly enforce it using a tool like `hindent`[10].

Apart from this, I have attempted to write idiomatic Haskell (and C). Especially in the `Codegen` module, I often take advantage of `do` notation and constructs like the `forM` function, which enables an almost imperative coding style on top of the code generation framework.

As mentioned in section 3.4, strings are limited to ASCII for all intents and purposes. This could be mitigated by converting Haskell `String`s to `ByteString`s before lowering them into LLVM IR, but that would break indexing as used by the string builtins because multi-byte characters would count as multiple characters without any special treatment. This is a minor issue in the context of a research project, so no fix was attempted so far.

Another small issue stems from how strings are compiled as detailed in section 3.5.2: currently, a sufficiently large array is allocated and each element is written to it sequentially. This leads to very verbose LLVM IR code compared to what Clang emits for string literals when compiling C: Because they are constant, they appear as constant global definitions in the IR, and accesses are carried out using the `GetElementPointer` instruction.

In section 5.3, I will discuss how to cut down on the excessive copying of values currently taking place in the generated code.

4.3 Challenges

Being new to compiler construction in general and the LLVM/Haskell toolstack, I had to overcome a few challenges.

[7]A version of the Quicksort test where a 10000-element input list was hardcoded also takes longer than a minute to compile. This shows that this issue is not specific to row literals.

[8]Which causes about 500 MB worth of logs to be written.

[9]See `https://github.com/tibbe/haskell-style-guide/blob/master/haskell-style.md`.

[10]See `https://github.com/chrisdone/hindent`.

Most notably, having only taken an introductory course to functional programming before, more advanced concepts like the state monad were unknown to me and had to be learned. Writing idiomatic Haskell and knowing when to introduce abstractions, which tends to hide but also introduce complexity, was somewhat hard for me. Dealing with low-level C and LLVM IR code was also something I was not used to before starting work on the KL to LLVM compiler.

Working with the research group-internal Cabal packages and third-party packages, most notably `llvm-general`, turned out to be somewhat error-prone as well, with Cabal frequently caching parts of old versions of packages and then throwing build errors. I'm not sure what exactly went wrong there, but deleting and recreating the Cabal sandbox always fixed it.

Something I have alluded to in section 2.6 is that the most recent version of `llvm-general` on Hackage only works with LLVM 3.5, which is outdated and no longer available on Homebrew since January. While building LLVM from source is of course possible, this requires more dependencies, is likely more error-prone and takes more time. However, the `llvm-general` GitHub repository contains a branch corresponding to LLVM 3.9, which is the most recent LLVM version at the time of writing. Following the submission of this thesis document, I intend to explore using this branch together with the Homebrew version of LLVM 3.9 instead.[11]

This might also fix another unexpected problem where, after installing seemingly unrelated updates on my laptop, the (previously working) option of running the LLVM optimization passes as part of the code generation stage started to result in segmentation faults during this stage of the compiler.

Also hard to debug were the unhelpful errors `llvm-general` throws when trying to serialize an erroneous AST to LLVM IR code.[12]

Ending this chapter on a positive note, I'm generally happy with how the KL to LLVM compiler turned out, especially its CLI and the output of status messages and intermediate results.[13] I'm also thankful for the help I received along the way in the form of the complete parser, typechecker and the weekly meetings, giving me an frequent opportunities to ask questions and request feedback.

[11]Another option is to switch to `llvm-hs`, a recent fork of `llvm-general` that "aims to improve the public release story, and better provide the interfaces needed for any Haskell project looking to leverage LLVM" (from `https://github.com/llvm-hs/llvm-hs/blob/llvm-4/README.md`). Additionally, its maintainers (including several high-profile Haskellers) are considering adding a LLVM IR pretty-printer that does not rely on the LLVM API to serialize the AST (see `https://github.com/llvm-hs/llvm-hs/issues/8`). I will keep an eye on `llvm-hs`

[12]See `https://github.com/bscarlet/llvm-general/issues/204`.

[13]Which haven't been in the spotlight in this document because they are unrelated to the compilation of KL to LLVM.

"Code is never finished, only abandoned."

— loosely based on Paul Valéry[1]

Chapter 5

Future Work

Even with the KL to LLVM compiler being able to emit working LLVM code that writes usable log files, there is still some work that can be done to improve and extend it. In this final chapter, I will discuss some potential improvements related to performance and interoperability, as well as additional features.

5.1 Data Structures

While there now exist fast versions of each data structure (array-backed lists and GHashTable-backed dicts and rows), further optimization is possible.

For example, the fast list implementation found in `KLToLLVM/code/cbits/arraylist.c` currently starts out with an array of size 10 and grows by a factor of 1.5 whenever it runs out of space – this might not be ideal. The growth factor could also be adjusted based on how many times the list has been grown already.

When duplicating arraylists using the `kltollvm_list_deepcopy()` function or even when compiling a large list literal, preallocating the space thus known to be required would decrease the number of memory (re-)allocations.

Hashing dictionaries and rows currently does not match their set-like semantics, instead treating them like lists of key-value pairs. For example, a dictionary `{'a': 1, 'b': 42}` currently hashes to a different value than `{'b': 42, 'a': 1}`. While this is not a problem with KL code generated by the SQL to KL compiler (because the element order tends to be consistent there), this behavior is incorrect and the hash functions should be modified accordingly.[2]

Code generation for list, dict and row literals currently emits an initialization step followed by a list of appends or updates (see section 3.8.1). Creating an array of the desired (keys and) values and passing this to an initialization function might make things slightly faster.

As mentioned in section 3.2, the Boehm GC cannot currently be combined with GLib. Remedying this would enable the use of these fast data structures without leaking memory, which is a prerequisite for executing large KL programs. The next section

[1] From `http://www.made2mentor.com/2008/10/code-is-never-finished-only-abandoned/`.

[2] Some notes on hashing sets of integers can be found here: `http://cstheory.stackexchange.com/q/3390`.

provides other approaches to solve the memory management complexities inherent to low-level C or LLVM IR code.

5.2 Memory Management

As can be seen when compiling KL programs with garbage collection disabled (see section 3.1.6), memory management is necessary to avoid memory leaks. Garbage collection is only one approach to solve this issue, and it incurs a certain runtime cost.

An alternative all-purpose approach to memory management is reference counting, which is natively supported by GLib as well (exemplified by the `g_hash_table_unref()` function explained in section 3.1.5). In simple terms, this involves storing the number of references to an object (i.e. a region of memory) alongside it, incrementing it whenever a pointer to the object is created or passed to a function and decrementing it when such pointers fall out of scope or are otherwise destroyed. Objects start out with a reference count of 1 in this context and are deallocated immediately – not only after the next heap scan, as is the case in a naive garbage collector – if their reference count drops to 0. Especially due to the native GLib support, this might be an approach worth considering for the KL to LLVM compiler.

Instead of relying on a garbage collector or reference counter to do the dirty work, a preprocessing pass could insert deallocation statements after the final occurrence of a variable. This is feasible due to KL's simplicity and the fact that it is side-effect free (apart from printing and log writing). My initial approach to exploring this would be a pass traversing the AST in reverse order (from bottom to top), inserting a `Free` statement (which would have to be added to the `Kernel`) before the first occurrence of each variable in the reversed AST. Loops and conditional statements somewhat complicate things here, for example in the case of a variable declared at the start of a function but last used inside a loop – a naive approach would free it in the first iteration of the loop. Deallocating variables only in the scope in which they were defined would fix this issue.

A topic that ties into memory management is the question of when to copy values that might be modified in the future, but whose original form is still required.

5.3 Copy on Write

For example, when constructing a non-atomic value purely for passing to a function or inserting into a data structure, some unnecessary work is done. Consider the following KL program, annotated with a high-level description of the LLVM IR code generated by the current version of the KL to LLVM compiler.

```
Kernel Language
1  # allocate memory for a pointer
2  var l : [int];
3
4  # allocate heap memory for an integer and store 1 (during code
   ↪  generation for the literal 1)
5  # same for 2
6  # create an empty list of integers
```

```
# dereference the previously created heap pointer and store the value
↪ pointed to (which is 1) at a fresh heap memory address (a pointer
↪ to which is then inserted into the list) to prevent future
↪ modifications to the source heap memory from affecting the value
↪ now saved in the list
# same for 2
l = [1, 2];

# create a deep copy the entire list while accessing the variable before
↪ passing the copy to the print builtin, which does not write to its
↪ argument
print(l)
```

As one can clearly see even in a simple example like this, the naive emulation of KL's pass-by-value semantics through copying values whenever they are duplicated leads to extraneous memory allocation and a lot of work for the garbage collector.

This degrades performance considerably – in some cases, by two orders of magnitude or more.[3] For a language primarily intended for data manipulation like KL, this is obviously less than ideal.

An approach where values are only copied when they will be written to in the future would help prevent extraneous memory allocations. As KL code is side-effect free (apart from printing and log writing, which will not pose any issues here), static analysis (i.e. traversal of the KL AST) based on simple rules (e.g. the fact that printing will not modify the value being printed) should be sufficient to discern values where copying is necessary from ones that can simply be passed along without change.

However, implementing such a copy-on-write approach might increase compile times. Researching how other compilers for high-level languages solve this issue would help in judging the effort required.

5.4 Interoperability and Scope

As of now, the KL to LLVM compiler takes a file containing KL code and emits LLVM IR code or an executable. When run, this executable writes log files that can then be processed further, eventually resulting in data provenance. This does the job, but it cannot be seamlessly integrated into the existing provenance derivation pipeline (see figure 2.1), instead relying on files as intermediary storage.

By using the optimization and just-in-time compilation (JIR) functionality offered by `llvm-general` and providing a library interface to the KL to LLVM compiler, the reliance on intermediary file story could be minimized. This could be combined with the switch to a newer LLVM version as outlined in section 4.3.

Another thing worth considering is reimplementing the second phase of the data provenance analysis pipeline to take advantage of the speed LLVM offers. This second phase concerns itself with actually deriving data provenance based on the logs and the input KL file and is currently the main bottleneck of the pipeline. In the future, considering different kinds of provenance (e.g. *how-provenance*, which deals with determining which code regions were involved in a computation) might also be an option.

[3] I can give this estimate because I only noticed this issue halfway through writing the compiler – most KL code generated by the SQL to KL compiler actually works correctly without all of this copying. Trying to compile nontrivial hand-written code surfaced the problem.

5.5 Miscellaneous

As mentioned in chapter 4, the KL to LLVM compiler sometimes takes several minutes during the code generation stage. It seems to get especially slow when a large number of literals is present in the KL code, as is the case for the Visiblity and Contour queries tested in section 4.1. Exploring what makes the code generator behave this way is likely necessary before beginning work on supporting the second phase of the provenance analysis pipeline.

Code generation for `for` loops currently is optimized for the linked list implementation, relying on the helper functions mentioned in section 3.1.6 to iterate through the list. Modifying the `for` loop code generator to take better advantage of the arraylist implementation's $\mathcal{O}(1)$ element access would make it slightly faster. Also, merging the two almost equivalent body blocks into one would make the emitted LLVM IR code somewhat shorter.

As previously mentioned in section 4.2, strings are effectively ASCII-only, which could potentially be fixed by using a C library that knows how to deal with unicode strings, e.g. `libiconv`.[4] The additional setup overhead and complexity might not be worth it, however.

In its present state, the KL to LLVM compiler cannot deal with the `null` value. This does not seem to be a pressing issue: I haven't seen this deficiency impact the compilation of KL code generated by the SQL to KL compiler. For this reason and because it would require significant modifications to how KL values are translated to LLVM values, this feature is not yet implemented. One possible approach is to model values as pointers to values (which is already the case for values of non-atomic types) and view null pointers as the `null` value.

To squeeze some extra performance out of the generated LLVM IR code, changing the optimization pass ordering is advisable. The LLVM Project maintains a "Performance Tips for Frontend authors"[5] pace on which it is noted that "one of the most common mistakes made by new language frontend projects is to use the existing [optimization] pipelines as is. These pass pipelines make a good starting point for an optimizing compiler for any language, but they have been carefully tuned for C and C++, not your target language. You will almost certainly need to use a custom pass order to achieve optimal performance."

[4]See `https://www.gnu.org/software/libiconv/`.

[5]See `http://llvm.org/docs/Frontend/PerformanceTips.html#pass-ordering`.

Bibliography

[BCR04] David F. Bacon, Perry Cheng, and V. T. Rajan. A unified theory of garbage collection. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, pages 50–68, New York, NY, USA, 2004. ACM. `http://www.cs.virginia.edu/~cs415/reading/bacon-garbage.pdf`.

[BKWC01] Peter Buneman, Sanjeev Khanna, and Tan Wang-Chiew. Why and Where: A Characterization of Data Provenance. In *Lecture Notes in Computer Science, Volume 1973, International Conference on Database Theory (ICDT 2001)*, pages 316–330. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001. `http://repository.upenn.edu/cgi/viewcontent.cgi?article=1209&context=cis_papers`.

[Cha] Oliver Charles. 24 Days of GHC Extensions. `https://ocharles.org.uk/blog/posts/2014-12-01-24-days-of-ghc-extensions.html`.

[Die] Stephen Diehl. Implementing a JIT Compiled Language with Haskell and LLVM. `http://www.stephendiehl.com/llvm/`.

[Ern08] Hartmut Ernst. *Grundkurs Informatik*, pages 469–470. Vieweg + Teubner Verlag, Wiesbaden, 2008.

[HHPJW07] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A History of Haskell: Being Lazy With Class. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages (HOPL III)*, San Diego, CA, USA, April 2007. `http://research.microsoft.com/en-us/um/people/simonpj/papers/history-of-haskell/history.pdf`.

[Ism15] Nadejda Ismailova. Development of a Data Provenance Analysis Tool for Python Bytecode. BSc thesis, Eberhard Karls Universität Tübingen, 2015. `http://db.inf.uni-tuebingen.de/attachments/thesis-ismailova-2015.pdf`.

[Kor10] Sandeep Koranne. Boehm GC: garbage collection. In *Handbook of Open Source Tools*, pages 151–154. Springer Science & Business Media, Berlin, Heidelberg, 2010. `https://books.google.de/books?id=ukXrNh2g6fQC&pg=PA151&redir_esc=y#v=onepage&q&f=false`.

[LA04] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo

Alto, California, Mar 2004. http://llvm.org/pubs/2004-01-30-CGO-LLVM.pdf.

[LLAS17] Chris Lattner, Casey Liss, Marco Arment, and John Siracusa. Accidental Tech Podcast. Episode 205. Podcast, 2017. http://atp.fm/205. Transcript available at http://atp.fm/205-chris-lattner-interview-transcript.

[Mü16] Tobias Müller. Have Your Cake and Eat it, Too: Data Provenance for Turing-Complete SQL Queries. In *Proceedings of the VLDB 2016 PhD Workshop*, New Delhi, India, September 2016. http://db.inf.uni-tuebingen.de/staticfiles/publications/cake-turing-2016.pdf.

[Piz] Filip Pizlo. Introducing Riptide: WebKit's Retreating Wavefront Concurrent Garbage Collector. https://webkit.org/blog/7122/.

Erklärung

Hiermit erkläre ich, dass ich diese schriftliche Abschlussarbeit selbstständig verfasst habe, keine anderen als die angegebenen Hilfsmittel und Quellen benutzt habe und alle wörtlich oder sinngemäß aus anderen Werken übernommenen Aussagen als solche gekennzeichnet habe.

Datum, Ort, Unterschrift