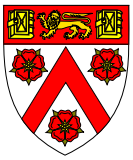
UNIVERSITY OF
CAMBRIDGE

Federico Stazi

Just-In-Time Compilation for dynamically typed languages

Part II Computer Science



Trinity College

This dissertation is submitted for Part II of the Computer Science Tripos in 2022

Declaration

I, Federico Stazi of Trinity College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose. I am content for my dissertation to be made available to the students and staff of the University.

Signed *Federico Stazi*
Date *May, 2022*

Proforma

Candidate Number:	**2426E**
Project Title:	**Just-In-Time Compilation for dynamically typed languages**
Examination:	**Computer Science Tripos - Part II, 2022**
Word Count:	**11996** [1]
Code Line Count:	**6791**[2] (+937 copied and modified from the SLANG interpreter)
Project Originator:	The dissertation author and Professor Alan Mycroft
Supervisor:	Professor Alan Mycroft

Original Aims of the Project

The design and implementation of *DLANG-VM*, a Just-In-Time compiling interpreter for DLANG, a minimalistic functional language, based on the SLANG language from the Compiler Construction course. DLANG-VM aims to be a simple, modular and efficient tool for students learning about these technologies and for researchers performing experiments. Core deliverables include the *DLANG-C* compiler to DLANG byte-code, the DLANG-VM byte-code interpreter and a testing framework.

Work Completed

All core deliverables were achieved in the early stages of the project, which was then expanded on multiple fronts and includes extensions such as multiple JIT compilation techniques, various JIT compiler optimisations and a garbage collector. The project also includes *Meta-DLANG-VM*, an alternative and separate implementation of DLANG-VM.

Special Difficulties

None.

[1] Calculated using `texcount`
[2] Calculated using `cloc`

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The first programming languages in history were developed as an additional layer of abstraction on top of the computer's machine-code. FORTRAN (created in 1957) uses a *compiler* to translate high-level code to machine-code [1], while Lisp (created in 1958) uses an *interpreter* to read and evaluate high-level code at run-time [2]. In the years that followed, new programming languages were developed, and they mostly used one of these two techniques, compilation or interpretation. As programs got more and more complex, their drawbacks emerged: with the former, source code is not portable, compilers can get complex and hard to maintain, and the programming languages are still very close to the machine level; for the latter, performance is significantly worse. This led, in the 1980s and 1990s, to the birth of *hybrid* languages such as Java, Perl and Python, involving a compilation stage to portable *byte-code* and interpretation of this byte-code. The newly introduced layer of abstraction of an intermediate language between high-level code and machine-code proved to be very successful in tackling the drawbacks of compilation and interpretation [3], and today these languages consistently rank among the top most widely used languages [4].

Hybrid languages have most of the advantages of interpreted languages and better speed, but cannot achieve the performance of compiled languages. Undoubtedly, execution speed is, in some circumstances, something programmers cannot compromise on. For this reason, research in this field has focused on trying to improve the performance of interpreters, and *Just-In-Time (JIT) compilation*, popularised by the developers of the Sun Java Virtual Machine in the late 1990s [5], became one of the most prominent and successful solutions. It has evolved over time and is today, in its many different variations, part of many modern interpreters, such as OpenJDK's and Oracle's Java Virtual Machines, the PyPy Python interpreter and the SpiderMonkey JavaScript interpreter.

1.1 Project Aim

The aim of the project is implementing a JIT compiling interpreter, *DLANG-VM*, for the minimalistic functional programming language *DLANG*, intended to be a **simple**, **modular** and **efficient** pedagogical tool for students learning about JIT compilation and for researchers interested in performing experiments on the topic.

Project summary. The project involves three main components:

1. ***DLANG-C***, a compiler from DLANG source code to DLANG byte-code;
2. ***DLANG-VM***, a virtual machine and JIT compiling interpreter of DLANG byte-code, and its alternative, *Meta-DLANG-VM*, which uses *Meta-Tracing JIT compilation*;
3. a **testing framework** to assess the correctness of the components and evaluate different JIT compilation techniques and optimisations.

1.2 Motivation

The project is motivated by the importance and interest in Just-In-Time compilation in both industry and academia: this technique is crucial for interpreted languages, which would otherwise be unable to compete with compiled languages in most circumstances. However, industrial-strength interpreters are such large and complicated projects that only small modifications can reasonably be achieved within the time limits of a Part II project. This led to the decision of working on a minimalistic language, with the overarching goal of creating a more approachable, yet realistic, interpreter for pedagogical and experimental purposes.

I designed **DLANG** and chose it as the minimalistic language for my project. It is a *dynamically typed* version of **SLANG**, the functional language developed by Professor Timothy G. Griffin for the Part IB Compiler Construction course [6]. In particular the DLANG-C byte-code compiler, which I created by modifying the SLANG compiler, compiles to (almost) the same *stack-based byte-code instructions* of SLANG, and the language is *fully backwards compatible* with it. DLANG is ideal for the task, as its simplicity makes it more approachable than industrial-strength interpreters, but its richness of features makes it a realistic language which can give useful insights on these techniques.

1.3 Related Work

Work in the literature on Just-In-Time compilation forms the theoretical foundations of the project. Cramer et al. wrote one of the earliest papers using the term in 1997, specifically for *Function JIT* compilation, discussing how it was applied to the Sun JVM [5] and building on work on run-time compilation from previous decades, such as that of Deutsch and Schiffman who applied the similar approach of *Incremental Compilation* to the Smalltalk-80 language in 1984 [7]. More recent pieces of work by Gal et al. propose an alternative JIT compilation technique, *Tracing JIT*, used in interpreters such as the HotpathVM JVM in 2006 [8] and the TraceMonkey JavaScript interpreter in 2009 [9]. Finally, in 2015, Bolz and Tratt wrote an assessment of the latest innovation in JIT compilation, *Meta-Tracing JIT*, used for the automatic creation of JIT compiling interpreters, and its impact on the development of interpreters [10].

As far as other components of interpreters and compilers are concerned, the literature provides a wide range of algorithms and techniques that can be applied to DLANG-VM. Optimisations provide a further reduction of the overheads of dynamically typed languages: those used in DLANG-VM's JIT compiler are the typical compiler optimisations discussed in the literature. Some of the most effective, which can be categorised as *Global Data Flow Analyses*, are *Available Expressions Analysis* [11] applied to run-time checks, and *Live Variable Analysis* [12]. Garbage collection is an important component of any virtual machine and was first defined in 1960 by McCarthy in the paper presenting the Lisp programming language (although it was originally called *Reclamation Cycle*) [2]. In the years that followed, many garbage collection algorithms were proposed: one of these, developed by Cheney [13], is the theoretical foundation for the garbage collector used in DLANG-VM.

Finally, a paper by Vandercammen et al. published in 2018 [14] has similar aims to those of this project. They note that "Designed and highly optimized for performance, [JIT compilation frameworks] are difficult to experiment with" and try to create a "minimalistic yet flexible" framework for studying JIT compilation. The key difference is that their framework is totally flexible in the language being interpreted, but more constrained in how this is JIT compiled; DLANG-VM enforces the interpretation of a specific language, DLANG, but gives greater flexibility in other aspects of JIT compilation and of the virtual machine. *The project aims to contribute to the field with a solution that is more approachable, especially for students, because it is ready to be used and does not require the design of a new language from scratch.*

Chapter 2

Preparation

This Chapter describes the techniques introduced in Chapter 1, how I applied them to the project and why I chose them. The first is the suite of compilers and interpreters for SLANG, the starting points for DLANG-C and for the design of the DLANG language (Section 2.1), followed by the theoretical foundations on Just-In-Time compilation (Section 2.2) and generation of machine-code at run-time (Section 2.3). Two more general aspects in the design of compilers and interpreters that also apply to DLANG-VM are then discussed: optimisations of JIT compiled code (Section 2.4) and algorithms for garbage collection (Section 2.5). The final Sections describe how correctness of the project is tested (Section 2.6), its requirements and how I prepared to efficiently achieve them (Section 2.7), the external tools, libraries and techniques I used (Section 2.8) and the starting point (Section 2.9).

2.1 The SLANG compiler and interpreter

SLANG is an implementation of the language described in the Part IB Semantics of Programming Languages course, and was developed by Professor Timothy G. Griffin for the Part IB Compiler Construction course [6]. *It is a functional language, with static typing, explicit type annotations and Call-By-Value semantics.* It also provides typical constructs of imperative languages, such as loops, references and I/O. It is implemented in OCaml.

The SLANG software package includes, in addition to source code interpreters and machine-code compilers, a compiler from source code to byte-code and a byte-code interpreter, as in the hybrid model of compilation and interpretation discussed in Chapter 1. The SLANG byte-code is a *stack-based instruction set*, as shown in the example program in Figure 2.1. Instructions operate on a *stack*, which contains values and pointers to complex objects on a *heap*. The package also includes several SLANG programs used for testing.

These features make SLANG a great starting point for the project. There are only 22 byte-code instructions, but these are enough for an expressive language that is comparable to other widely used functional languages such as ML (albeit without its powerful type inference system). The existence of a simple interpreter simplifies testing and the existence of a compiler removes the burden of implementing one from scratch. Modifying the SLANG compiler was relatively straightforward, and gave me the chance to focus on the most interesting aspects of the project early on.

```
1 PUSH STACK_INT 2
2 MK_CLOSURE L0 0
3 APPLY
4 HALT
5 LABEL L0
6 LOOKUP STACK_LOCATION -2
7 PUSH STACK_INT 1
8 OPER ADD
9 RETURN
```

Figure 2.1: SLANG byte-code of: `(fun (x : int) -> x + 1 end) 2`

2.2 Just-In-Time Compilation

As mentioned in Chapter 1, Just-In-Time compilation involves an interpreter compiling byte-code to machine-code at run-time. The rationale behind it is simple, but the effects are remarkable: *compiled instructions improve performance by reducing interpreter execution during the interpretation of the program*, skipping some fetch and decode steps of the interpreter's fetch-decode-execute cycle. JIT compilation operates on *sections*, sequences of byte-code instructions of the program, as follows:

1. The interpreter keeps track of how often program sections are interpreted.
2. Based on the information from step 1, if the interpreter infers that a certain section of code will be used so often in the future that it would benefit from JIT compilation (taking into account the factors discussed below), then it JIT compiles it.
3. When the interpreter reaches a JIT compiled program section, it runs the faster JIT compiled machine-code instead of interpreting the instructions.

There is a fundamental trade-off that must be taken into account when JIT compiling code. Sections of code are, in general, faster after being JIT compiled, but there are some drawbacks:

- compilation and bookkeeping have time overheads, which are only amortised by faster execution if the section being JIT compiled is used frequently in the program;
- JIT compiled code has a memory overhead for its storage;
- JIT compiled code must interact seamlessly with interpreted code, and this introduces additional time overheads.

The policies determining which sections of code are most likely to benefit from JIT compilation generally depend on the system and the use case. However, JIT compiling interpreters can be classified based on which sections of code are considered for JIT compilation: these categories are *Function JIT*, *Tracing JIT* and *Meta-Tracing JIT*, and are described below.

2.2.1 Function JIT

Function JIT is the original and simplest form of JIT compilation. The interpreter keeps track of how often each function is called, and compiles the whole function if it is called frequently. This technique limits potential issues arising from the interactions of interpreted and JIT compiled code, because functions are a mostly isolated set of instructions. For example, a JIT compiled function can assume that no other interpreted instruction will interact with items on its stack frame. The benefits of Function JIT are described in greater detail in Cramer's et al. paper about how it was being pioneered for the JIT compilation of Java byte-code on the JVM [5].

2.2.2 Tracing JIT

Tracing JIT is an alternative and more modern approach, which emerged due to the necessity of better memory and power efficiency for JIT compilers. Some of the early work on this mentions the efficiency issues of Function JIT: "Sun's Java HotSpot Virtual Machine 1.4.2 for PowerPC includes a just-in-time compiler that achieves an impressive speedup of over 1500% compared to pure interpretation. However, this comes at the price of a total VM size of approximately 7MB, of which about 90% can be attributed to the just-in-time compiler." [8].

With Tracing JIT, JIT compilation can be applied to any sequence of instructions; these sequences are called *traces*. It requires more bookkeeping than Function JIT, but is much more flexible: it can, for example, compile small portions of functions (such as loop bodies) or sequences of non-contiguous branching instructions, including those spanning multiple function calls. To deal with branching instructions, *guards* are used: these are special instructions placed after each branch, to check if execution is following the *expected* path in a JIT compiled trace. If a guard check fails when executing a trace, execution is halted and control returns to the interpreter. The example in Figure 2.2 shows a trace extracted from a loop body, with guards placed in such a way that control can go back to the interpreter if an *unexpected* branch is taken. This example benefits particularly from Tracing JIT compared to Function JIT.

```
1 def f(n):
2   return 2 * n if n != 12 else 0
3
4 def g(n):
5   return n * n if n != 34 else 0
6
7 s = 0
8 for i in range(100):
9   s += f(i) * g(i)
```

```
branch i != 12
guard
t1 = 2 * i
branch i != 34
guard
t2 = i * i
s += t1 * t2
```

Figure 2.2: The *expected* trace of a loop body with two `guard` statements

Different interpreters, such as PyPy [15] or the Java HotpathVM [8], use a similar approach to choose which traces are JIT compiled: choosing the *loops* in the program, because any loop is also a trace. This is an approximation, as some traces may not be loops. To find a loop that is a good target for JIT compilation, these track *hot loop headers*, defined as the instructions that are a frequent destination of backward branches. The algorithm has two modes of operation:

- the efficient *profiling mode*, which counts the number of times each loop header is reached by backward branches, and changes to tracing mode when one exceeds a certain threshold;
- the expensive but rare *tracing mode*, which tracks all instructions being executed, JIT compiles the sequence when it returns to the loop header or discards it if too long, and then changes back to profiling mode.

This finds frequently executed traces with high probability and minimal bookkeeping overhead.

2.2.3 Meta-Tracing JIT

The latest and most innovative approach to Just-In-Time compilation is Meta-Tracing JIT, which involves two simple steps for the creation of a JIT compiling interpreter. The first is implementing the interpreter in RPython (a slightly reduced Python dialect), without any code explicitly performing JIT compilation and only a few additional code annotations (explained in Section 3.4.1). The second is compiling it with the RPython compiler, which produces the JIT compiling interpreter.

This is clearly a revolutionary approach, *as the burden of implementing JIT compilation for the interpreter is fully automated*, but is less known and used. As described in one of the earliest papers on this topic, "The RPython language allows tracing JIT VMs to be automatically created from an interpreter, changing the economics of VM implementation" [10]. The drawback of this approach is, as with most solutions automating burdensome tasks in computing, performance. An automatically generated Meta-Tracing interpreter is usually outperformed by an ad-hoc and heavily optimised JIT compiling interpreter [10].

2.3 Machine-code Generation

Generating machine-code at run-time is a *fundamental technical requirement for JIT compilation.* Doing this is not straightforward, and doing it in an efficient, portable, maintainable and safe way is even harder: I therefore use an Open-Source C library for this, *GNU Lightning* [16].

Efficiency The library only applies a few basic optimisations and transformations to the code, making compilation faster, and leaving more room for higher-level optimisations.

Portability and Maintainability The library abstracts away most low-level and platform-dependent aspects of machine-code, *exposing a simple interface that greatly simplifies the complex problem of machine-code generation.* It has three caller-saved registers, three callee-saved registers and an orthogonal RISC-style instruction set with arbitrarily sized immediates. However, the project makes a stronger assumption on the machine, motivated in Section 3.2.1, by assuming the existence of at least five callee-saved registers (compatible with x86-64 machines).

Safety On Linux, the library generates executable code at run-time by combining calls to the `mmap` and `mprotect` system calls using the `PROT_EXEC` flag, to respectively write the sequence of instructions to memory and mark it as executable. These are the same system calls used when loading dynamic libraries, and do not require the deactivation of *Executable-Space Protection* (the technique uses the NX bit of memory regions of the program stack to protect against exploits of bugs in regular programs, such as *Buffer Overflow* and its modern variants, but does not stop programs *intentionally* marking a region of memory as executable).

2.4 Compiler Optimisations

Optimisations are algorithms applied as intermediate steps of compilation, with the goal of producing a faster and more efficient program. They are generally applied to intermediate representations of code, such as *GIMPLE* for the GCC C++ compiler [17]. The intermediate instruction representation is usually a *Three Address Representation*: instructions take at most three arguments, of which at most two are the input and at most one is the output. These powerful algorithms for compiled languages can also be applied to JIT compiling interpreters, but code transformations and optimisations must occur at run-time. *Optimisations are crucial because they reduce one of the most significant overheads for JIT compiling interpreters of dynamically typed languages: run-time checks. Their use makes JIT compilation even more effective.*

To define and implement optimisations on a program P with instructions $p_1, p_2, p_3, ..., p_N$ in an intermediate language, it is useful to represent P as a *Flow Graph*. Optimisations can operate on the graph, or on its *Basic Blocks* and *Traces*. Their definitions follow, made under the assumption that the instruction set has conditional branches with fixed destination and no unconditional jumps. Figure 2.3 is an example of a Flow Graph, its Basic Blocks and a Trace.

Definition 2.4.1 (Flow Graph)**.** The Flow Graph $G = (V, E)$ of P is a *simple directed graph*, whose nodes V and edges E are given by:

$$V = \{p_1, p_2, p_3, ..., p_N\} \qquad E = \{(p_i, p_j) \mid p_j \text{ may follow } p_i \text{ in the execution of } P\}$$

Node p_N has no outgoing edges, therefore G is a line (for any $1 \leq i < N$, $(p_i, p_{i+1}) \in E$) with additional edges for branches (if $j \neq i + 1$ and $(p_i, p_j) \in E$, p_i is a branch with destination j).

Definition 2.4.2 (Basic Block)**.** A Basic Block B of a Flow Graph G is a sequence of unique nodes $b_1, b_2, b_3, ..., b_M$ such that:

- For any $1 \leq i < M$, (b_i, b_{i+1}) is the one and only outgoing edge of b_i;
- For any $1 < i \leq M$, (b_{i-1}, b_i) is the one and only incoming edge of b_i.

It follows that a Basic Block is a sub-sequence of contiguous instructions in P such that:

- No instruction, except for the first one, can be a branch destination;
- No instruction, except for the last one, can be a branch;

These properties make Basic Blocks great targets for optimisations.

Definition 2.4.3 (Trace)**.** A Trace T of a Flow Graph G is a sequence of nodes $t_1, t_2, t_3, ..., t_M$ such that for all $1 \leq i < M$ there exists an edge $(t_i, t_{i+1}) \in E$. Traces are more general than Basic Blocks: there are no constraints on the other edges of the nodes or on their position in P.

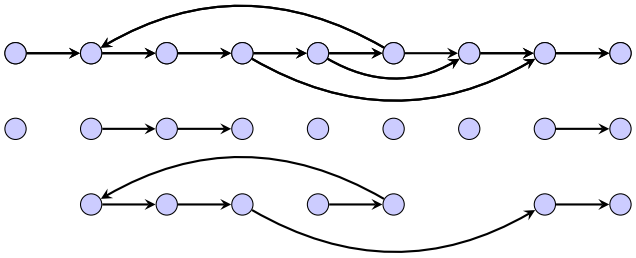


Figure 2.3: An example of a Flow Graph, its Basic Blocks and a Trace

In general, optimisations first analyse the program, and then apply modifications based on the analysis results. *Live Variable Analysis* can be used for *Dead Code Elimination*, while *Available Expressions Analysis*, among other purposes, can be used for the reduction of run-time checks (this is an ad-hoc algorithm for DLANG-VM, and is explained in Section 3.3.2).

Definition 2.4.4 (Live Variable Analysis)**.** A variable x is *semantically live* at a certain node n of a Flow Graph G if there is a possible execution path from n whose observable effects are influenced by the value of x. Proving semantic liveness is undecidable, but there exists a less tight decidable definition: a variable x is *syntactically live* at node n if there is a path in G from n to a node n', and n' takes x as input. This is an overestimation: if a variable is semantically live, then it is also syntactically live, but the opposite is not true.

For a node n of G, let $succ(n)$ be the destinations of n's outgoing edges, $write(n)$ the variables written by n and $read(n)$ the variables read by n. The least fixed-point of this set of equations determines the live sets of all nodes in G:

$$live(n) = \left(\bigcup_{s \in succ(n)} live(s) \right) \setminus write(n) \cup read(n)$$

Definition 2.4.5 (Dead Code Elimination)**.** If the only effect of an instruction n is to write to a variable x, and $x \notin \bigcup_{s \in succ(n)} live(s)$, then instruction n can be eliminated, because the value of x is not used later in the program. When applying this algorithm, all side effects of the instruction must be considered: for example, instructions that assign to x a user input, or the result of a division whose divisor may be 0, cannot be eliminated.

Definition 2.4.6 (Available Expressions Analysis)**.** An expression e is *semantically available* at a node n of G if e is computed in all possible execution paths leading to n. As with liveness, semantic availability is undecidable, but *syntactical availability* is a decidable overestimation.

For a node n of G, let $pred(n)$ be the sources of n's incoming edges, $kill(n)$ the expressions containing a variable modified by n and $gen(n)$ the expressions computed by n. The greatest fixed-point of this set of equations determines the available sets of all nodes in G.

$$avail(n) = \left(\bigcap_{p \in pred(n)} avail(p) \cup gen(p) \setminus kill(p) \right)$$

2.5 Garbage Collection

Garbage collection is a method used by most virtual machines to automatically reclaim memory that will no longer be used, called *garbage*, ensuring that the program does not unnecessarily run out of memory. Garbage must be defined in a safe way: the garbage collector must never reclaim memory that the program will use in the future, but it may preserve memory that will not be used in the future. This is not ideal, but sometimes required due to undecidability.

Definition 2.5.1 (Garbage)**.** On a machine with no registers, a stack and a heap:

- items on the stack *are not* garbage;
- items referenced by items that are not garbage *are not* garbage;
- all other items *are* garbage.

As described in the assessment on the topic by Bacon et al., two opposing techniques exist for garbage collection, *Reference Counting* and *Tracing* [1] [18]. The former, used by the C++ *Smart Pointers* [19], keeps for each item on the heap a *counter* of the number of references from other items: if it gets to zero, it becomes garbage (this follows from Definition 2.5.1) and can be collected, thus decreasing the reference counters of the items it references (the opposite is not true: an item with non-zero counter may be garbage because of cycles). The latter, used by many Java Virtual Machines, recursively marks non-garbage items starting from the stack, and may do so concurrently with the program execution or by stopping it.

[1] Not to be confused with Tracing JIT.

Definition 2.5.2 (Cheney's Algorithm)**.** Cheney's Algorithm is a Tracing garbage collection algorithm, and works as follows [13] (Figure 2.4 shows the steps on a simple example):

1. recursively mark items that are not garbage starting from the stack;
2. copy marked items to a new heap;
3. adjust the pointers on the stack and on the heap.

All steps have linear time complexity in the number of non-garbage items on the stack and heap. To achieve this, recursion in step 1 stops on already marked items to avoid visiting them again.

After collection, the heap is *compact*: all items are stored contiguously. This makes the otherwise complicated choice, especially in JIT code, of where to allocate new items on the heap, trivial.

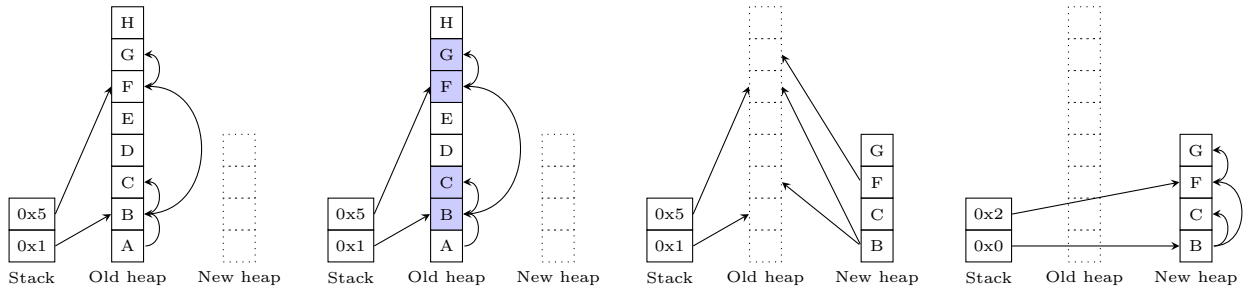


Figure 2.4: The steps of Cheney's algorithm applied to a simple example

2.6 Correctness Testing

Absence of bugs is a very important but, in some sense, impossible goal for compilers and interpreters [20]. The approach proposed by Hoare in 1969 is *Verification*, which uses axioms and inference rules to prove properties of programs [21]. The alternative approach, used in this project, is *Validation*, which is limited to empirically showing that the program behaves correctly on some inputs (in the case of interpreters, that some input programs are interpreted correctly).

When validating an interpreter, especially with the additional complexities of JIT compilation and optimisations, having numerous and effective tests is crucial. For this reason, three kinds of tests are included: *Unit Tests*, *Differential Tests* and *Fuzz Tests*. How these are implemented and applied to the project is discussed in Section 3.5 and their results are evaluated in Section 4.2.

Unit Testing Unit Testing is used to assess the correctness of individual and isolated components of a program. It is particularly useful and effective when testing data-structures used in various parts of the program, but relies on the ability of the test designer to successfully detect errors caused by less frequent and hard to predict corner-cases.

Differential Testing (or Output Comparison) Differential Testing, formalised by McKeeman in 1998 [22], is a form of *Integration Testing* (it assesses the correctness of entire programs) often applied to interpreters (or compilers). It consists of executing the same program on two or more different interpreters and comparing the outputs. The reasoning behind the approach is described by Sheridan, who calls it *Output Comparison*, in a paper about tests for a C99 compiler: "different compilers are likely to have different bugs, and so bugs in one compiler are likely to result in output which differs from another compiler's for the same input" [23].

Fuzz Testing (or Fuzzing) Fuzz Testing is a form of automated testing, which involves two steps: generating random inputs and executing the software being tested on such inputs. It is particularly useful for compilers and interpreters because:

- programming languages are often defined as a *Context Free Grammar (CFG)*, and this greatly facilitates the generation of random input programs provoking desirable behaviours;
- there is no need to manually check test results, since Differential Testing can also be applied to randomly generated input programs.

The motivation for choosing Fuzz Testing is straightforward: *it is a fully automated process with minimal marginal cost, and minimal human bias.* In a 1984 paper from Duran and Ntafos, they state that Fuzz Testing "[...] can be cost effective for many programs [...] allows one to obtain sound reliability estimates [...] can discover some relatively subtle errors without a great deal of effort [...] provide[s] a very high segment and branch coverage" [24]. More recently, Fuzz Tests were shown to be more effective than other forms of testing with controlled experiments on human subjects by Ceccato et al. [25], and the OSS-Fuzz project by Google "has found over 30,000 bugs in 500 open source projects" (as of June 2021) [26]. More advanced forms of Fuzzing exist, such as the *Mutation-based* and *Coverage-based* variants, but are not explored in this project. Section 4.2 describes the positive impact of Fuzzing on the project.

2.7 Requirements Analysis

The Success Criteria of the project are included in the Project Proposal. As an overview, the three main required components, also introduced in Section 1.1, are:

- **DLANG-C**: the compiler from DLANG source code to byte-code;
- **DLANG-VM**: the virtual machine and JIT compiling interpreter of DLANG byte-code;
- **tests**: to assess the correctness and performance of the project.

For each component, I identified a set of deliverables and their priorities in Table 2.1. I used this Table and the PERT Chart [27] in Figure 2.5 as tools for the detailed planning of the project timeline, and the Gantt Chart [28] in Figure 2.6 to track its progress.

Component	ID	Deliverable	Priority
DLANG	DC-comp	DLANG-C compiler	Succ. Crit.
	DC-annot	Removing DLANG explicit type annotations	Low
DLANG-VM	DVM-int	Interpreter for DLANG byte-code	Succ. Crit.
	JIT-func	JIT compiler using Function JIT	Succ. Crit.
	JIT-trac	JIT compiler using Tracing JIT	High
	JIT-meta	JIT compiler using Meta-Tracing JIT	Low
	DVM-pol1	Basic policy to choose sections to JIT compile	Succ. Crit.
	DVM-pol2	Other policies to choose sections to JIT compile	High
	DVM-mem1	Memory allocation at run-time	Medium
	DVM-mem2	Garbage collection	Low
	DVM-opt	Optimisations for JIT compiled machine-code	Low
tests	CORR	Testing the correctness of the project	Succ. Crit.
	PERF-jit1	Testing the performance of JIT compiled code	High
	PERF-jit2	Testing the performance of different JIT compilation techniques	Medium
	PERF-opt	Testing the performance of different JIT compiler optimisations	Low

Table 2.1: Deliverables and their priorities

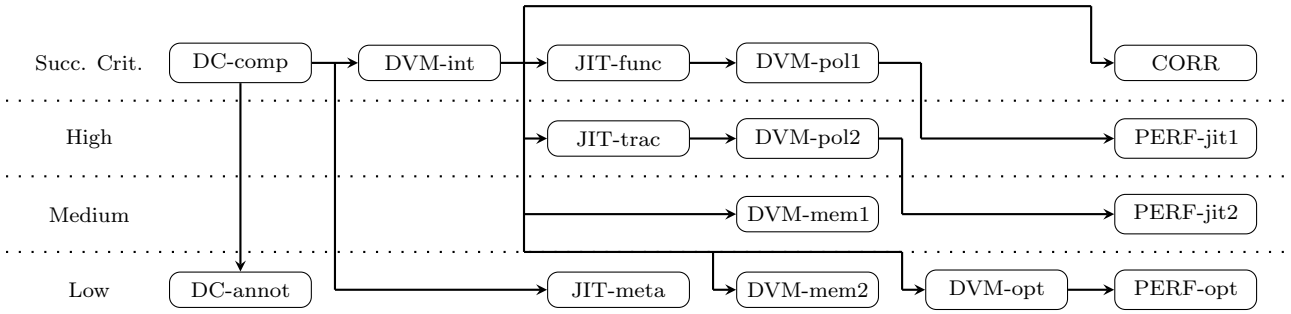


Figure 2.5: The PERT Chart showing deliverables' dependencies

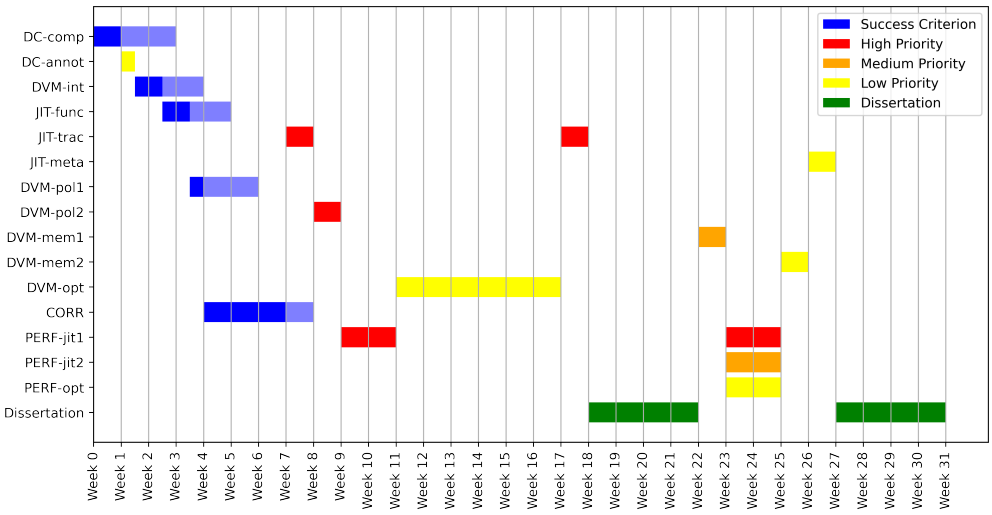


Figure 2.6: The Gantt Chart tracking project progress

2.8 Software Engineering Tools and Techniques

This Section explores the most important aspects of my approach to the development of the project, from both the organisational and technical points of view.

2.8.1 Software Development Process

For the project, I used a *Spiral Model* of Software Development [29]. On a per-deliverable basis:

1. I identified a goal for the current deliverable (a basic initial implementation for new deliverables or an improvement for existing ones).
2. I identified risks and expected completion time.
3. I developed and thoroughly tested the code.
4. I decided, based on the priorities in Table 2.1 and dependencies in Figure 2.5, whether to start working on a new deliverable or a new goal for the current deliverable.

On a smaller scale, an example of how I used the Spiral Model is the implementation of JIT compilation for sequences of byte-code instructions. I first implemented JIT compilation for individual instructions, then for sequences without branches, then for functions, and finally for traces, with each iteration building on top of the tested work from the previous iteration.

On a larger scale, I used the Spiral Model to implement machine-code optimisations. I started with an initial iteration of ad-hoc data-structures and algorithms to remove redundant run-time checks in JIT compiled code. This proved to be effective, but also gave useful insights on how it could be generalised for other optimisations. I therefore re-iterated the process by modifying the code from the previous iteration, obtaining a general framework for arbitrary optimisations.

2.8.2 Tools and Libraries

A list of the most important tools and libraries used in the project is in Table 2.2, together with information on why I chose each of these components. An important choice worth highlighting is using C++ (and CMake) for DLANG-VM, instead of other Object-Oriented languages such as Java or functional languages such as OCaml, for its performance, imperative style and lower level of abstraction. Another is GNU Lightning, chosen for its simplicity compared to similar alternatives such as LLVM and libJIT. All the tools in Table 2.2 (with the exception of GitHub), are Open-Source, and this was a key aspect I took into consideration when choosing the tools (which is not mentioned in the Table to avoid repetitions).

2.8.3 Version Control and External Dependencies

As shown in Table 2.2, I use Git and GitHub for version control, with additional backups to Google Drive. As far as external dependencies are concerned, I use Git Submodules to include the SLANG repository: this is added as a subdirectory to the project and is automatically cloned when cloning the project, but always references to the actual SLANG repository. GNU Lightning and GoogleTest are instead downloaded and compiled automatically by the CMake script during the build process.

2.8.4 Licensing

Table 2.2 also includes the licence for each tool and library: I listed them all, understood their terms and ensured they are compatible with my project and its publication on GitHub as a public repository. These are all, to varying degrees, permissive licences, and allow for private and commercial use, with the conditions of the code and possible modifications of the libraries being disclosed. In addition to that, I contacted Professor Griffin to ensure he is aware of my modifications to the SLANG compiler. I chose an MIT License for my project, which is permissive and compatible with the licences of all libraries used, and all files include a copyright notice in the first line. The project is publicly available on GitHub.

2.9 Starting Point

I had some experience working with C++, and knowledge limited to Part IA and Part IB courses for Python and OCaml. In part IB, I successfully ran and inspected the SLANG compiler as an exercise for the Compiler Construction course, thus giving me a head start on the implementation of the DLANG-C compiler. I learnt about all other topics from the literature while working on the project.

As far as existing code is concerned, DLANG-C is a modified version of the SLANG compiler from the Part IB Compiler Construction course: Section 3.1 describes which parts I removed and modified to make the compiler work for DLANG. As described in Section 3.5.2, some of the test inputs were also part of the SLANG repository. The modifications to the SLANG compiler and all new code that is part of this project were written after the start of Michaelmas Term.

Tool/Library	Used for:	Chosen because:	Licence:
Programming Languages			
C++	DLANG-VM	fast, scalable, OOP	
Python3	various scripts	quick development iterations	
OCaml	DLANG-C	used in SLANG	
RPython	Meta-DLANG-VM	provides Meta-Tracing JIT	
Build Tools and Compilers			
GCC	C++ compilation	standard in most Linux OSs	GPL-3.0+
CMake	C++ build tool	platform-independent	BSD-3-Cl
CPython	Python compilation	standard Python implementation	PSF-2.0
RPython compiler	RPython compilation	official RPython implementation	MIT
OCaml compiler	OCaml compilation	official OCaml implementation	LGPL-2.1
OCamlbuild	OCaml build tool	simple, used in SLANG	GPL-2.0
External Libraries and Components			
Boost	C++ options parsing	powerful library	BSL-1.0
GNU lightning	C++ JIT compilation	simple, platform-independent	LGPL-3.0+
GoogleTest	C++ Unit Tests	powerful and accessible	BSD-3-Cl
Dharma	Fuzz Tests generation	accessible and CFG-based	MPL-2.0
ocamllex	OCaml lexing	part of the OCaml tool-chain	LGPL-2.1
ocamlyacc	OCaml parsing	part of the OCaml tool-chain	LGPL-2.1
Development Tools			
GDB	C++ debugging	official GNU debugger	GPL-3.0
Perf	C++ profiling	part of the Linux Kernel	GPL-2.0
Cpplint	C++ linter	adheres to Google's style guide	BSD-3-Cl
Pylint	Python linter	adheres to the PEP8 style guide	GPL-2.0+
Git	version control	simple, powerful	GPL-2.0
GitHub	source code hosting	free	

Table 2.2: The most important tools used in the project

2.10 Summary

The preparation for the project, before writing any of the code, involved many of its different aspects. First, I familiarised myself with SLANG, the starting point of the project, and with the literature on JIT compilation, machine-code generation, compiler optimisations and garbage collection. After this, I set out the testing techniques for the project, and listed requirements and external dependencies, ensuring the compatibility of all licences. Finally, I prepared a detailed project plan, and started the implementation.

Chapter 3

Implementation

This Chapter discusses the implementation of the two steps in the workflow for the execution of a DLANG program: the **compilation** of high-level functional-style DLANG code to DLANG byte-code, performed by the *DLANG-C* byte-code compiler (Section 3.1) and the **interpretation** of DLANG byte-code, performed by *DLANG-VM* (Section 3.2). It also describes DLANG-VM's API for modular extensions (Section 3.3) and an alternative to DLANG-VM, *Meta-DLANG-VM*, a Meta-Tracing JIT interpreter (Section 3.4). The final Sections discuss how correctness and performance are tested (Section 3.5) and show the structure of the repository (Section 3.6).

3.1 DLANG-C compiler

This Section describes the DLANG-C compiler, and highlights key differences between DLANG and SLANG (Section 2.1): as discussed in Section 2.9, most of the code for DLANG-C is taken from the SLANG compiler, and the differences between the languages are reflected in their compilers. Both are implemented using OCaml.

Figure 3.1 shows the intermediate representations transforming source code to byte-code and Appendix A includes the transformations of an example program.

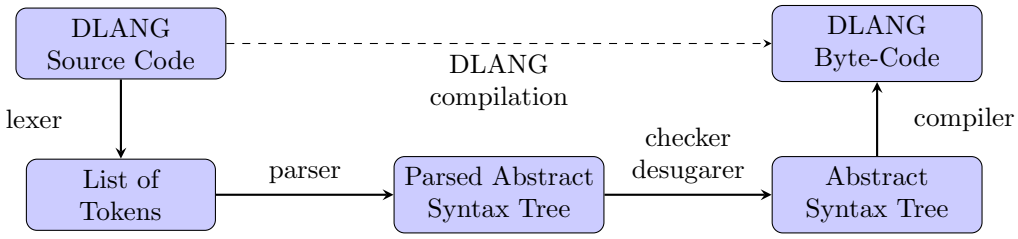


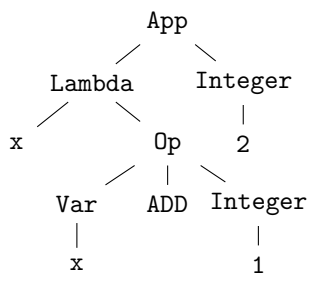
Figure 3.1: The intermediate representations in the DLANG-C compiler

DLANG Source Code is the high-level functional language given as input to the DLANG compiler. As shown in the example program in Figure 3.2, it resembles other functional languages such as OCaml and ML, with a few *basic data types* which can be composed into *complex data types*, and expressions such as `if-then-else` statements, `while` loops, `let` bindings, lambda abstractions and imperative-style sequencing. The only difference from SLANG is that DLANG, because of dynamic typing, does not require type hints, such as the ones included in the SLANG example in Figure 2.1. These can still be used, thus ensuring *full backward compatibility*, but are ignored. The list of expressions as a CFG can be found in Appendix B.

A **List of Tokens** is produced by the lexer generated by the `ocamllex` library and then the **Parsed Abstract Syntax Tree** is produced by the parser generated by the `ocamlyacc` library. These OCaml libraries have benefits in terms of implementation time and correctness.

The **Abstract Syntax Tree** is the result of compile-time *static analysis* on the Parsed Abstract Syntax Tree. *DLANG is a dynamically typed language*, therefore a relatively small number of transformations occur at this stage. The most important difference between the SLANG and DLANG compilers is that the former performs static type checks, while the latter does not, meaning that *a successfully interpreted DLANG program may fail compilation in SLANG.* The only static check in DLANG-C ensures that no variable is used out of scope. Additionally, the code is *desugared*: the syntactic sugar in the Parsed AST is removed in the AST. For example, the `let x = n in e end` expression becomes `(fun x -> e) n`. Figure 3.2 includes an AST.

DLANG Byte-Code is generated as the final step of the pipeline by a recursive transformation on the AST. There is a minor difference between SLANG and DLANG in the use of the `LABEL` instruction. The instruction is used in SLANG for branches implementing `if-then-else` statements, `while` loops, `case` statements and function bodies. DLANG uses `LABEL` for all these, except function bodies, for which `FUNCTION` is used instead. The instructions are otherwise identical, but their explicit distinction makes detecting functions in DLANG-VM much easier, which is useful for JIT compilation and optimisations. Figure 3.2 shows an example of DLANG byte-code, including `FUNCTION`. Appendix B contains the list of DLANG byte-code instructions.




```
1 PUSH STACK_INT 2
2 MK_CLOSURE L0 0
3 APPLY
4 HALT
5 FUNCTION L0
6 LOOKUP STACK_LOCATION -2
7 PUSH STACK_INT 1
8 OPER ADD
9 RETURN
```

Figure 3.2: Abstract Syntax Tree and DLANG byte-code of: `(fun x -> x + 1 end) 2`

3.2 DLANG-VM

DLANG-VM is the core component of the project. It is a virtual machine and interpreter of DLANG byte-code, implemented in C++. This Section introduces the basic functionalities of DLANG-VM: the model of computation, the intermediate representations, the fetch-decode-execute cycle and JIT compilation. Section 3.3 follows, and shows how modular customisations can be built on top of these foundations to extend and improve DLANG-VM.

3.2.1 The Model of Computation

A *Model of Computation* is the definition of the interface between software and hardware in a computer system: it defines the available instructions for the software and how the hardware executes them. The same concept can be applied to a virtual machine and the byte-code instructions it interprets. In general, when the model greatly differs from the real machine model, the instruction set becomes simpler, but performance degrades, making optimisations even more important. *The computational model of DLANG-VM is that of a Stack Machine*, similar to those of the Java Virtual Machine and the CPython byte-code interpreter.

Definition 3.2.1 (Stack Machine)**.** In a Stack Machine, data is stored on a *stack*. Instructions read from and write to the stack, and their arguments are the elements on the top. For example, a program computing $1 + 1$ has instructions: `PUSH 1, PUSH 1, ADD`. The `ADD` operation has no arguments, because the two integers on the top of the stack are *implicit arguments*.

Specifically, DLANG-VM has a stack of basic objects and a *heap* of complex objects. Additionally, five registers are used: `cp` points to the current instruction, `sp` and `fp` define the top and bottom of the current stack frame, `hp` points to the next available location on the heap and `tmp` stores temporaries for JIT compilation. The stack frame of a function includes the argument, a pointer to its closure, the `fp` of the caller, the return address and the temporary values of the function.

Items on the stack and heap are represented as *(value, tag)* pairs, with the tags in Appendix B. The stack contains simple values, including heap pointers, while the heap contains both simple values and *headers* for their composition into complex objects. For example, Figure 3.3 shows how a pair is implemented.

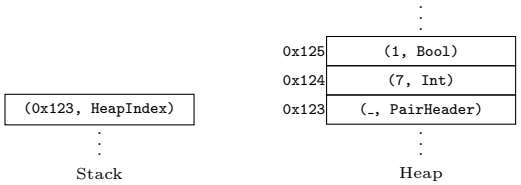


Figure 3.3: `(7, True)` pair on the stack top

The stack and heap are implemented by the `Memory` class: this is similar to `std::dynarray`, a dynamically allocated and fixed size container proposed for the C++ Standard Library in 2013 but never incorporated [30]. The size of items is 64 bits, 32 for the value and 32 for the tag.

3.2.2 Internal code representations

The byte-code input to DLANG-VM undergoes some transformations at run-time into intermediate representations, each designed for a specific goal, as shown in Figure 3.4. *I designed these additional layers of abstraction to ensure that the best possible interface is available to each component of DLANG-VM, simplifying the implementation and comprehension of the algorithms.* Appendix A includes the transformations of an example program.

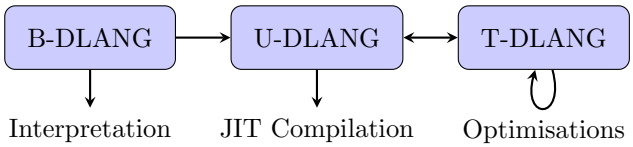


Figure 3.4: The internal representations of code in DLANG-VM

B-DLANG is the internal representation of DLANG byte-code (*B-* stands for byte-code). The language is used for basic interpretation, when no JIT compilation occurs. B-DLANG is stack-based: since implicit arguments on the stack are not the ideal representation for JIT compilation and optimisations, U-DLANG and T-DLANG are used for these instead.

U-DLANG is a set of lower-level instructions (*U-* stands for micro), with most instructions being implemented by one or two machine-code instructions. These are closer to the level of RISC instructions, but abstract away some of their complications (e.g. `UMove` uses `movr` or `movi` according to the type of its argument). Their arguments are immediates, registers and memory locations, making the language ideal for JIT compilation. When JIT compiling B-DLANG instructions, these are first converted to U-DLANG and then compiled. *U-DLANG instructions bridge the gap between stack-based B-DLANG and machine-code by moving away from the stack-based representation while hiding machine-code behind a simpler interface.*

T-DLANG is a *Three Address Representation* (*T-* stands for three) of U-DLANG instructions (they map 1-to-1), taking variables and immediates as arguments. When optimising and JIT compiling B-DLANG code, it is converted to U-DLANG, then to T-DLANG, optimised, converted back to U-DLANG and finally JIT compiled. *T-DLANG instructions offer an additional layer of abstraction over U-DLANG instructions by hiding multiple low-level representations of arguments behind simple variables*, making it a great instruction set for optimisations.

Each instruction is implemented as an object of a subclass of `BInstruction`, `UInstruction` or `TInstruction`. `BInstruction` objects have an `interpret` method modifying the VM state, and a `getUInstructions` method for conversion to U-DLANG. `UInstruction` objects have a `jitCompile` method emitting machine-code instructions (as explained in Section 3.2.4) and a `getTInstruction` method for conversion to T-DLANG. `TInstruction` objects have methods facilitating optimisations, and a `getUInstruction` method for conversion to U-DLANG.

3.2.2.1 U-DLANG and T-DLANG arguments

The arguments to U-DLANG and T-DLANG instructions are organised in the hierarchy of subclasses of `UArgument` and `TArgument` respectively, as shown in the *UML Diagrams* in Figure 3.5. U-DLANG instructions take `UOperand`, `ULocation`, `UImmediate` or `URegister` arguments, which are internally implemented by subclasses as shown in the diagram.

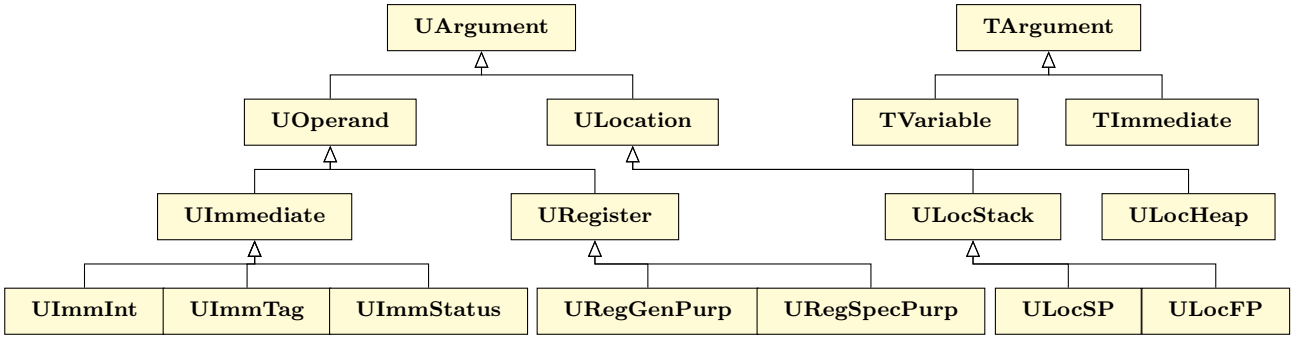


Figure 3.5: The UML Diagrams of `UArgument` and `TArgument` subclasses

There is a simple 1-to-1 correspondence between U-DLANG and T-DLANG, shown in Table 3.1. The complex part of the transformation, is that of the arguments, which occurs as follows:

- A `UImmediate` becomes a `TImmediate`.
- A `URegister` becomes a `TVariable`, the same register is represented by the same variable.
- A `ULocHeap` becomes a `TVariable`, but the same memory location may be represented by multiple variables, which is inevitable because heap addresses are determined at run-time.
- A `ULocStack` becomes a `TVariable`, and the same memory location is always represented by the same variable when compiling functions (but not traces): this is possible because, in functions, instructions change the stack size by fixed offsets.

From these definitions, it follows that optimisations on T-DLANG code can always assume that *two identical variables always represent the same entity (register or memory location).*

3.2.2.2 U-DLANG and T-DLANG instructions

Table 3.1 shows the list of all U-DLANG and T-DLANG instructions. U-DLANG arguments are written `l` for memory locations, `r` for registers and `p` for registers or immediates. All T-DLANG arguments can be variables or immediates and are named `x`. An important characteristic, useful for optimisations, is the presence of instructions for run-time checks: `TAG-CHECK` and `MEM-CHECK`. Some instructions have additional constant parameters not mentioned in the Table (the operator for `UNARY` and `OPER`, the destination for `GOTO` and `BRANCH` and the expected tag for `TAG-CHECK`).

U-DLANG	T-DLANG	Description

`GET r l`
`SET l r`
`MOVE r p`
	`MOVE` x_1 x_2	Copies a value from memory to a register (get), from a register to memory (set) or from a register/immediate to a register (move); all these become `MOVE` in T-DLANG

`UNARY r p`	`UNARY` x_1 x_2	Performs a unary operation
`OPER r` p_1 p_2	`OPER` x_1 x_2 x_3	Performs a binary operation
`LABEL`	`LABEL`	Marks the start of a B-DLANG instruction
`GUARD`	`GUARD`	Exits the compiled code if the `cp` is not as expected
`TAG-CHECK l`	`TAG-CHECK x`	Throws an error if the tag is not as expected
`MEM-CHECK l`	`MEM-CHECK x`	Throws an error if the location is out of memory bounds
`APPLY`	`APPLY`	Marks the end of a B-DLANG `APPLY` instruction
`RETURN`	`RETURN`	Marks the end of a B-DLANG `RETURN` instruction
`HALT`	`HALT`	Stops the program
`GOTO`	`GOTO`	Jumps unconditionally to the destination[1]
`BRANCH p`	`BRANCH x`	Jumps to the destination if the argument is non-zero

Table 3.1: The instructions in T-DLANG and U-DLANG

3.2.3 Fetch-Decode-Execute loop

Figure 3.6 shows the implementation of the Fetch-Decode-Execute loop in DLANG-VM (lines for the verbose mode and tracking of execution statistics are omitted). The code performs the code transformations described in Section 3.2.2 and uses the extensions API of Section 3.3:

- Line 3 invokes garbage collection.
- Line 8 attempts JIT compilation if no code section starting here was compiled before.
- Line 10 invokes the JIT compilation policy to determine if a code section must be compiled.
- Lines 14 to 17 convert B-DLANG code to U-DLANG.
- Lines 20 to 25 optimise U-DLANG code using T-DLANG internally.
- Lines 28 to 31 JIT compile the U-DLANG instructions.
- Lines 34 to 38 terminate JIT compilation and store the generated function.
- Lines 43 to 52 run the compiled code or interpret the instruction.

Additionally, lines 6 and 44 notify the JIT compilation policy on the execution.

3.2.4 Just-In-Time Compilation

The GNU Lighting library is used to generate machine-code at run-time, as introduced in Section 2.3. It is a C library: as such, it requires C++ wrappers for compatibility with DLANG-VM, and to match the OOP style of the project. Several classes are used to integrate the library into the rest of the DLANG-VM code, which is very different in style and level of abstraction.

[1]When the Flow Graph is constructed, this instruction has an edge to its destination, but also to the next instruction, for consistency with the Flow Graph properties described in Section 2.4.

```
while (vm_->status == VirtualMachine::Running) {
  // Run Garbage Collection
  memoryManager_->collectGarbage(vm_);

  // Count landings for this instruction
  jitPolicy_->notifyLanding(vm_->cp);

  if (!compiled_[vm_->cp]) {
    // Get the jit compilation policy
    auto jitSequence = jitPolicy_->makeJITSequence(code_, vm_->cp);

    if (!jitSequence.isEmpty()) {
      // Create u-code instructions
      Code<UInstruction> uCode;
      for (auto cp : jitSequence.getCps()) {
        uCode += code_.getInstruction(cp)->makeUInstructions(vm_);
      }

      // Optimize u-code
      Code<UInstruction> uCodeOptimized;
      if (jitSequence.isFunction()) {
        uCodeOptimized = optimizationsSequence_->optimizeFunction(uCode);
      } else {
        uCodeOptimized = optimizationsSequence_->optimizeTrace(uCode);
      }

      // Compile u-code while keeping jit state
      auto jit = std::make_shared<JITState>(jitSequence, vm_);
      for (auto uInstruction : uCodeOptimized) {
        uInstruction->jitCompile(vm_, jit);
      }

      // Store the pointer for the main and all secondary entry points
      compiled_[vm_->cp] =
          std::make_shared<CompiledInstructions>(jit->compile(vm_));
      for (const auto& [cp, size] : jitSequence.getEntryPoints()) {
        compiled_[cp] = compiled_[vm_->cp];
      }
    }
  }

  // Interpret the instruction or run its compiled code
  if (compiled_[vm_->cp]) {
    jitPolicy_->notifyRunJIT(vm_->cp);
    compiled_[vm_->cp]->run();
  } else {
    try {
      code_.getInstruction(vm_->cp)->interpret(vm_);
    } catch (const RuntimeError&) {
      vm_->status = VirtualMachine::Status::RuntimeError;
    }
  }
}
```

Figure 3.6: Fetch-Decode-Execute loop in DLANG-VM

JIT compilation in GNU Lightning is very simple: after initialising the state, a user calls `jit_prolog()`, followed by a sequence of library method calls generating machine-code instructions, and finally `jit_emit()`, which compiles the instructions and returns a function pointer. Examples of calls generating instructions are included in Figure 3.7.

Since, as any C library, GNU Lightning does not use classes and objects, it does not adhere to the *Resource Acquisition Is Initialization (RAII)* principle. This shortcoming is solved by the `JITState` class, which initialises the state required for compilation in its constructor and releases it in its destructor, automating an otherwise manual and error-prone process. Additionally, this:

- saves callee-saved registers and uses them for the `cp`, `sp`, `fp`, `hp` and `tmp` pointers;
- creates branches at the start to all possible entry points for the code section;
- provides methods to add labels and branches in the code (both if the branch precedes or succeeds the label, as the library has different methods for the two cases).

After a `JITState` object is created, regular GNU Lightning methods can be called, followed by a call to its `compile` method to produce a `CompiledInstructions` object, a wrapper for a C function pointer to the JIT compiled code. Figure 3.7 shows how the class can be used.

The `JIT` class is a static class that can be inherited by any class using GNU Lightning methods. Without it, library methods require a `jit_state_t` object storing the compilation context, which must be named `_jit` and always be in scope, as the library uses `#define` directives to implicitly pass it as an argument to all methods. By inheriting from `JIT`, the variable is automatically added to the scope, and method calls in the class work without explicitly passing the object.

The `JITVM` class simplifies the implementation of JIT instructions. For example `JITVM::call` calls an external C++ function from JIT compiled code (e.g. to request memory allocation). To do so, it temporarily stores caller-saved registers, passes the arguments, calls the method, stores the result in a register and restores caller-saved registers. A *template parameter pack* and *fold expression* are used to handle any number of arguments. Figure 3.7 shows other utilities.

Altogether, these classes provide an interface simplifying JIT compilation with minimal overhead.

```
MOVI R0 0
s:
ADDI R0 R0 1
BNEI R0 9 s
```

```
// cps: the compiled sequence
// cp: the branch destination
JITState jit(cps, ...);
jit_movi(JITVM::r0, 0);
jit_addi(JITVM::r0,
         JITVM::r0, 1);
jit->addDirectBranch(
  jit_bnei(JITVM::r0, 9), cp);
return jit->compile();
```

```
auto _jit = jit_new_state();
jit_prolog();
jit_movi(JIT_R0, 0);
auto s = jit_label();
jit_addi(JIT_R0, JIT_R0, 1);
jit_patch_at(
  jit_bnei(JIT_R0, 9), s);
return jit_emit();
```

Figure 3.7: RISC code in C++ using GNU Lightning with and without DLANG-VM interfaces

3.3 DLANG-VM modular extensions

As stated in Chapter 1, the purpose of DLANG-VM is to be an interpreter that students interested in learning about JIT compilation can easily understand, and a tool that researchers performing experiments can easily modify. To achieve this, I designed DLANG-VM to be an individual core component on top of which *modular and high-level extensions, with minimal dependencies on DLANG-VM and no dependencies on each other*, can be implemented. This results in the narrow interfaces introduced in Figure 3.6 and the possibility of extending DLANG-VM with new JIT compilation policies, T-DLANG optimisations and memory management.

The UML diagram in Figure 3.8 shows how the *Strategy Design Pattern* is applied to the `DlangVM` class: it contains objects that define its strategy and are determined at run-time. In addition to that, since in general multiple optimisations can be applied, the `OptimizationsSequence` class uses an approach similar to that of the *Composite Design Pattern* to internally compose many optimisations with a single method call.

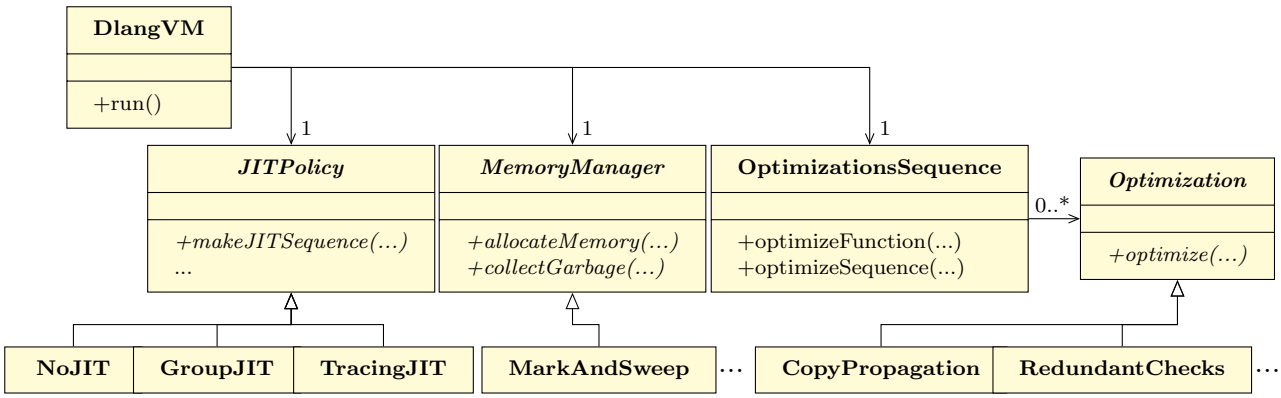


Figure 3.8: The UML Diagram of `DlangVM` and the subclasses of its modular extensions

3.3.1 Just-In-Time Compilation Policies

The JIT compilation policies are implemented as subclasses of the `JITPolicy` class, which creates `JITSequence` objects via the `makeJITSequence` method (Figure 3.8). The `JITSequence` object can be empty, meaning that JIT compilation is not performed, or contain a sequence of code pointers describing which section of code is JIT compiled: *this a flexible solution that allows for any JIT compilation policy to be implemented.* As shown in the fetch-decode-execute loop in Figure 3.6, the object receives updates on the execution via the `notifyLanding` and `notifyRunJIT` methods, information that can be stored internally and used to determine which sections to JIT compile. *DLANG-VM implements three `JITPolicy` classes:*

`NoJIT` always returns an empty sequence, and is used when no code is ever JIT compiled.

`GroupJIT<group>` is a template-class whose argument determines what is JIT compiled: `Individual` (a single B-DLANG instruction), `Block` (a sequence of B-DLANG instructions with no branches) or `Function`. The first two are useful for testing purposes, while the second one is the implementation of Function JIT (Section 2.2.1) for DLANG-VM. To simplify the definition of new groups, the `BGroupRole` class can be used to define the role of an instruction in the group (for example, the B-DLANG `BReturn` instruction is a subclass of `EndOf<Function>`).

`TracingJIT` implements Tracing JIT compilation, and closely follows the definition of the algorithm described in Section 2.2.2. It has constructor parameters to modify the threshold for JIT compilation (the number of times a loop header must be reached before it is considered to be *hot*) and the maximum trace size.

3.3.2 Just-In-Time Compilation Optimisations

Optimisations in DLANG-VM, introduced in Section 2.4, are transformations applied to a U-DLANG code section to obtain a better performing U-DLANG code section with identical observable behaviour. Let U and T be the sets of U-DLANG and T-DLANG code sections respectively, let $t : U \to T$ be the (bijective) transformation of U-DLANG code to T-DLANG code, and let $o_1, o_2, o_3, \dots o_n : T \to T$ be a sequence of n (possibly repeated) optimisations. When optimised, a section of U-DLANG code $u \in U$ is transformed into $u' \in U$ as follows:

$$u' = t^{-1}\left(o_n\left(\dots\left(o_3\left(o_2\left(o_1\left(t\left(u\right)\right)\right)\right)\right)\dots\right)\right)$$

This design *exploits the benefits of T-DLANG* described in Section 3.2.2 by using it as the intermediate representation, and *provides a very simple but flexible interface.* The user implements optimisations as high-level $T \to T$ transformations, and then arranges them into the sequence $o_1, o_2, o_3, \dots o_n$; DLANG-VM implements t, t^{-1} and then composes these transformations to obtain the whole optimisation process. *This makes optimisations modular components that are easy to modify because of their independence and easy to compose because of their flexibility.*

Transformations are implemented as subclasses of `Optimization` implementing the virtual method `TCode optimize(TCode)`, and are composed into an `OptimizationsSequence` as shown in the UML diagram in Figure 3.8. DLANG-VM provides an API simplifying the implementation of optimisations at many levels: at the single T-DLANG instructions level, at the Flow Graph level (for optimisations of entire functions or traces) and at the Basic Block level (for optimisations relying on the absence of branches). The `OptimizationError` exception can be used when the section of code being optimised contains instructions that surely fail: in this case, control gets back to the interpreter and the instructions are interpreted, so that the failing instruction is reported correctly.

DLANG-VM implements four optimisations at different levels. The most effective, as shown is Section 4.3.2, is ***Redundant Checks Elimination***, which uses *Available Expressions Analysis* (defined in Section 2.4) to remove unnecessary `TAG-CHECK` and `MEM-CHECK` instructions: a check instruction is removed if the same check is *available* at its node. It is a simplified version of the *Common Subexpressions Elimination* algorithm, because it uses availability but only applies it to checks, and can be categorised as a Flow Graph optimisation.

Another optimisation at the Flow Graph level is ***Dead Code Elimination***, which uses *Live Variable Analysis* (both defined in Section 2.4). Figure 3.9 shows the loop generating the live sets of the instructions by finding the least fixed point of the equation in Section 2.4. Two measures are taken to improve performance: nodes are iterated over in reverse order, since the live set of a node depends on the live sets of its successors, and the live sets are represented efficiently by a class making use of *bit-masks* internally.

Two more optimisations are implemented in DLANG-VM. At the level of Basic Blocks, ***Copy Propagation*** propagates the effects of `MOVE` operations whose second argument is an immediate: for example `MOVE a 1; OPER ADD b a 2` becomes `MOVE a 1; OPER ADD b 1 2` . It does not remove instructions, but it can improve other optimisations, as shown in Section 4.3.2. At the level of individual instructions, ***Constant Folding*** simplifies instructions whose result is known at compile-time: for example, `OPER ADD a 1 1` becomes `MOVE a 2`, and the check `TAG-CHECK-INT int` is removed.

```
do {
  liveHasChanged = false;
  for (const auto& node : nodes) {
    auto liveOld = live[node];
    live[node] = {};
    for (auto succ : node->getSuccessors()) {
      live[node].insert(live[succ]);
    }
    for (const auto& arg : node->getValue()->getWriteArgs()) {
      if (auto var = std::dynamic_pointer_cast<TVariable>(arg)) {
        live[node].erase(var->getUID());
      }
    }
    for (const auto& arg : node->getValue()->getReadArgs()) {
      if (auto var = std::dynamic_pointer_cast<TVariable>(arg)) {
        live[node].insert(var->getUID());
      }
    }
    liveHasChanged |= liveOld != live[node];
  }
} while (liveHasChanged);
```

Figure 3.9: Implementation of the loop generating the Live Sets in a Flow Graph

3.3.3 Memory Management

The policies for memory management introduced in Section 2.5 are implemented as subclasses of the `MemoryManager` class (Figure 3.8), and have two goals: allocating memory when the stack or heap are overflowing and performing garbage collection on the heap.

Memory allocation is implemented by the **`allocateMemory`** method, called by DLANG-VM when trying to access memory outside of current boundaries, possibly from JIT compiled code.

Garbage collection is implemented by the **`collectGarbage`** method, called by DLANG-VM at each iteration of the fetch-decode-execute loop, as shown in Figure 3.6. Collection cycles should occur rarely, therefore the method is expected not to do anything most of the times. Occasionally, it can modify the virtual machine state by performing a collection cycle.

DLANG-VM implements three memory managers. `NoAllocation` provides a fixed amount of memory and never allocates more. `AmortizedAllocation` allocates memory dynamically with constant amortised cost. `MarkAndSweepGC`, whose implementation of `collectGarbage` is shown in Figure 3.10, implements *Cheney's Algorithm* (Section 2.5) with relatively high-level code:

- Line 3 determines when a collection cycle starts.
- Lines 6 to 11 mark the non-garbage objects on the heap with a recursive function.
- Lines 14 to 17 copy and compact the heap.
- Line 20 adjusts the pointers to the new item locations on the heap.

```
1  virtual void collectGarbage(VirtualMachinePtr vm) {
2    // Use a policy to determine if collection starts
3    if (...) { return; }
4
5    // Mark phase
6    for (int idx = 0; idx < vm->sp; idx++) {
7      if (vm->stack[idx].tag == HeapIndex ||
8          vm->stack[idx].tag == HeapRef) {
9        markRecursive(vm->stack[idx].value, vm);
10     }
11   }
12
13   // Sweep phase - creating a new compressed heap
14   Memory newHeap(vm->heap.size());
15   for (size_t idx : marked_) {...}
16   vm->heap = newHeap;
17   vm->hp = marked_.size();
18
19   // Sweep phase - adjusting indices to the new heap
20   ...
21 }
```

Figure 3.10: Implementation of the `collectGarbage` method of `MarkAndSweepGC`

3.4 Meta-DLANG-VM

As introduced in Section 2.2.3, *Meta-Tracing JIT* is an alternative approach for JIT compilation involving the implementation of a simple interpreter in a Python dialect. Meta-DLANG-VM is the Meta-Tracing JIT interpreter for DLANG byte-code: as in DLANG-VM, it interprets byte-code programs produced by DLANG-C. This Section explains how Meta-DLANG-VM successfully implements this less known and innovative approach to JIT compilation.

3.4.1 RPython interpreters

The first step for the creation of a Meta-Tracing JIT compiling interpreter is implementing the interpreter in *RPython*. This is a slightly limited dialect of Python2, which simplifies static analysis: for example, variables cannot change type, run-time definitions of functions are not allowed and some libraries cannot be used (more details on the RPython Documentation [31]). Better static analysis enables better optimisations, with great performance benefits.

Next, various annotations for the JIT compiler must be added to the program. First of all, a call to `jit_merge_point` must be placed at the start of the fetch-decode-execute loop, and a call to `can_enter_jit` must be placed at the end of all backward branching instructions. Then, all variables in the loop must be assigned to one of two sets: *green* variables and *red* variables. Green variables are invariant in each loop iteration, and are used to identify which instruction is currently being executed (e.g. the code pointer). Red variables are the variables that can change during the loop execution (e.g. the machine state).

Finally, the interpreter is compiled with the `--opt=jit` option, used to automatically add JIT compilation to the interpreter. Compiling the interpreter takes a few minutes, but *the output produced is surprisingly powerful*: a fully functioning JIT compiling interpreter, with automatic memory management using a customised version of a *Generational Garbage Collector*. The JIT compiler automatically uses `jit_merge_point` to detect the start of the loop, `can_enter_jit` to detect backward branches, and the set of green variables to identify instructions and traces [15].

3.4.2 Interpreting DLANG Byte-Code

Implementing Meta-DLANG-VM is a straightforward application of the steps described in Section 3.4.1. The code pointer (`cp`) and the code itself are the *green* variables, while the state of the virtual machine is the *red* variable. The state is represented at a higher level of abstraction compared to DLANG-VM, simplifying the script even further. `sp` and `hp` are not needed: the

stack is a Python list, and items are appended or popped from it, while the heap is made implicit using Python object references. DLANG items can be values (integers in RPython), heap pointers (references in RPython) or heap headers for complex objects (lists of references in RPython). Parts of the implementations of the `VirtualMachine` and `Item` classes are shown in Figure 3.11. Variable types cannot change in RPython, therefore each item has a value, a pointer and a body (for headers), but only one is used.

```
class VirtualMachine:
  def __init__(self):
    self.stack = []
    self.cp = 0
    self.fp = 0
    self.status = 0

  ...
```

```
class Item:
  def __init__(self, tag=Tag.Unknown,
               val=0, ptr=None, bdy=[]):
    self.tag = tag
    self.val = val
    self.ptr = ptr
    self.bdy = bdy

  ...
```

Figure 3.11: Implementations of `VirtualMachine` and `Item`

3.5 Testing

This Section describes how the project was tested throughout the development process, and how the techniques described in Section 2.6 are applied to the *Validation* of DLANG-VM. Section 3.5.4 describes a different form of testing, Performance Testing. The repository includes a Python3 script, `test.py`, which runs all tests and can be used to test future extensions of the project: *easy-to-use tests will simplify the implementation of new extensions, as it simplified the implementation of those that are currently part of DLANG-VM.*

3.5.1 Unit Testing

The aim of Unit Testing is the assessment of the correctness of individual components of a program (Section 2.6). Their structure reflects the Object-Oriented structure of DLANG-VM: tests are grouped by class, and a single test creates an object of this class and inspects the results of method calls on the object. They are designed to test both typical behaviours, and also less common inputs that are harder to test at higher levels. Tests are implemented with the GoogleTest library for C++ Unit Testing, which simplifies their definition and automatically generates an executable performing all tests.

Unit Testing in DLANG-VM is applied to generic data structures used throughout the code: `Code<T>`, `FlowGraph<T>`, `Memory` and `IntSet`. There are a total of 21 Unit Tests.

3.5.2 Differential Testing

Differential Testing consists of executing the same program on multiple interpreters, and comparing their outputs (Section 2.6). The inputs to these tests are:

- all the 67 SLANG test programs [6] (valid because of DLANG's backward compatibility), which are also tested on the SLANG interpreter, assumed to be the *gold standard*;
- 35 additional DLANG programs, testing in particular run-time errors, which are also compared to their given expected output

Inputs are tested on DLANG-VM with many combinations of its options (270 in total: 2 JIT thresholds, 5 JIT policies, 9 optimisation sequences and 3 memory managers) and on Meta-DLANG-VM. For each input listed in `tests.json`, the `tests.py` script compiles it to byte-code, executes it on all applicable interpreters and compares the outputs.

3.5.3 Fuzz Testing

Fuzz Testing is applied to DLANG-VM as a variation to Differential Testing (Section 2.6): input DLANG programs are randomly generated and tested on DLANG-VM with the 270 combinations of its options and on Meta-DLANG-VM. Inputs are generated randomly using Dharma [32] from the definition of valid DLANG programs in the *Context Free Grammar* form (the CFG of DLANG is in Appendix B). Variable names are constrained to always be in scope, as they otherwise fail static checks at compile-time, and this constraint is applied by `test.py`.

I performed a sanity check by plotting the total number of byte-code instructions and the number of executed byte-code instructions of a set of 10,000 randomly generated input programs (excluding six non-terminating programs), shown in Figure 3.12. The plot shows a group of very short programs, a group of longer programs whose execution is of more than twenty instructions, and a larger group of quickly terminating long programs. To ensure that all kinds of programs are tested, some inputs are discarded so that *each group is represented by a third of the inputs tested.*

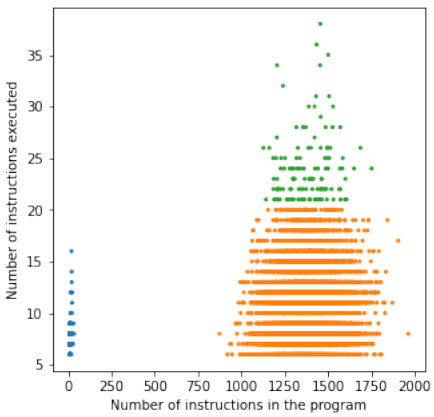


Figure 3.12: The plot of the total and executed number of instructions

This means that a good portion of tests are executed on "interesting inputs", but also that other inputs are not completely discarded. I ran a total of 15,000 tests on the final version of the project, 5,000 for each of the three groups, and all succeeded. However, during the development process, various bugs were found using the technique: these are listed in Section 4.2.

3.5.4 Performance Testing

Properly testing the performance of a program is a complicated task, made even harder by the limited time and hardware resources for this project. It is however of crucial importance for a project such as DLANG-VM, and I therefore implemented tests with two goals: reducing measurement errors as much as possible, and precisely measuring the uncertainty in the results.

Reducing measurement errors. When running a program, especially on a typical general-purpose laptop, many factors can influence execution in non-deterministic ways that are hard to detect and prevent. I therefore took some countermeasures to limit these effects:

- I used the `taskset` Linux tool to set the *processor affinity* of the process being benchmarked: this ensures that the program never performs expensive CPU migrations.
- I used the `nice` Linux tool to increase the scheduling priority of the process.
- When comparing two or more different interpreters, I interleaved their execution to ensure that any event on the system interfered more evenly with all of them.

Measuring uncertainty. To measure uncertainty and assess the quality of the results, the 95% confidence interval of all measurements is included. For n measurements $X_1, X_2, X_3, \ldots, X_n$, a 95% confidence interval can be defined as follows (a proof is included in Appendix D):

$$\overline{X}_n = \frac{1}{n} \cdot \sum_{i=1}^{n} X_i \qquad S_n^2 = \frac{1}{n-1} \cdot \sum_{i=1}^{n} \left(X_i - \overline{X}_n\right)^2$$

$$\left[\overline{X}_n - z_{0.025} \cdot S_n \cdot \sqrt{n}, \overline{X}_n + z_{0.025} \cdot S_n \cdot \sqrt{n}\right] \qquad z_{0.025} \approx 1.96$$

Performance evaluations in Section 4.3 are performed by the `test.py` script, which uses the techniques described in this Section to reduce errors and measure uncertainty.

3.6 Repository overview

Table 3.2 shows the repository of the project. The structure loosely reflects that of the Sections in Chapter 3. **`dlang_c/`** is the implementation of the DLANG-C compiler to byte-code (Section 3.1). **`dlang_vm/`** is the implementation of DLANG-VM, including core components (Section 3.2), modular extensions (Section 3.3) and Unit Tests (Section 3.5.1). **`meta_dlang_vm.py`** is the RPython Meta-Tracing interpreter (Section 3.4) and **`tests/`** contains scripts and files for testing (Section 3.5). The repository also includes SLANG as a Git Submodule in **`cc_cl_cam_ac_uk/`**.

For each tool, Appendix C shows the full list of command-line arguments. The `README.md` file contains the installation instructions for all components of the project.

File/Folder	Description	LOCs
`cc_cl_cam_ac_uk/`	Repository of the Compiler Construction course	
`dlang_c/`	DLANG-C byte-code compiler	
├─ `lexer/, parser/, checker/, compiler/`	Internal transformations (Section 3.1)	876[1]
├─ `front_end/, utils/, dlang_c.ml`	Command line parsing and other utilities	61[1]
`dlang_vm/`	DLANG-VM JIT compiling interpreter	
├─ `src/`		
│ ├─ `b_dlang/, u_dlang/, t_dlang/`	Internal representations (Section 3.2.2)	2809
│ ├─ `jit/, virtual_machine/, dlang_vm/`	Core components and features (Section 3.2)	736
│ ├─ `jit_policies/, optimizations/, memory_managers/`	Modular extensions (Section 3.3)	687
│ ├─ `data_structures/, options/, out/, main.cpp`	Command line parsing and other utilities	499
├─ `tests/`	Unit tests (Section 3.5.1)	265
├─ `external/`	CMake scripts downloading and installing libraries	37
├─ `CMakeLists.txt`	Main CMake build script for the project	10
`tests/`	Testing framework (Section 3.5)	
├─ `test.py, benchmark.py, tests.json, inputs/, dlang.dg`	Testing scripts, test inputs and CFG of the language	1116
`meta_dlang_vm.py`	Meta-DLANG-VM interpreter (Section 3.4)	405
`build.py`	Script for build automation	154
`LICENSE, README.md`	Licence and instructions on how to use the project	73

Table 3.2: The most important files and folders in the repository

[1]Most of these lines are copied or modified from the SLANG compiler

Chapter 4

Evaluation

This Chapter analyses the project to understand whether it accomplishes the desired goals. As stated in Chapter 1, the project is intended to be a **simple**, **modular** and **efficient** tool for learning about and experimenting with JIT compilation. For this reason, it must be evaluated in terms of what it provides and the measures taken to ensure that the code is of good quality for other users to interact with (Section 4.1), how it is validated to minimise errors (Section 4.2) and its performance (Section 4.3).

4.1 Project Success

The project achieves all Success Criteria and additional deliverables. The **DLANG-C** byte-code compiler is implemented (DC-comp, Success Criterion) and it supports programs both with and without explicit type annotations (DC-annot). The interpreter in **DLANG-VM** is also implemented (DVM-int, Success Criterion), and it supports Function JIT (JIT-func, Success Criterion) and Tracing JIT (JIT-trac). It implements various policies for JIT compilation (DVM-pol1, DVM-pol2), multiple optimisations (DVM-opt) and memory managers (DVM-mem1, DVM-mem2). An RPython Meta-Tracing JIT interpreter, Meta-DLANG-VM, is also implemented (JIT-meta). Finally the project has extensive **testing**, for correctness (CORR, Success Criterion) and for the performance of basic JIT compiled code (PERF-jit1), JIT compilation techniques (PERF-jit2) and optimisations (PERF-opt) (discussed in Section 4.3).

Some of these deliverables, such as Meta-Tracing JIT compilation (Section 2.2.2) and Fuzz Testing (Section 2.6), are *powerful but rarely known techniques that this project implements in a simple and approachable way.*

The project is designed to be modular and high-level in its extensions, but at the same time flexible, resulting in algorithms implemented with short and simple code (as shown in Chapter 3). The interfaces in DLANG-VM are meant to *help the expression of intent in code* and *encapsulate the implementation details*, which should not inhibit the implementation of new algorithms.

The project adheres to strict style guidelines: the Google C++ Style Guide [33], the PEP8 Python Style Guide [34] and the OCaml Programming Guidelines [35], enhancing readability and maintainability. All lines of code have a length of at most 80 characters and, in particular for C++ code, all method declarations and definitions are in separate files, modern C++ Standard Library components are used throughout the code, such as *Smart Pointers* and `optional<T>`, and classes follow the *Resource Acquisition Is Initialization* principle. Finally, scripts are used in many parts of the project for the automation of burdensome development and testing tasks. All components are platform independent and have simple interfaces, as those shown in Figure 4.1 and Appendix C, helping users quickly build, test and modify the project.

```
1 ./build.py all          2 ./tests/test.py fuzz

3 ./dlang_c/dlang_c fib.dlang -o fib.out

4 ./dlang_vm/dlang_vm --jit-policy tracing --verbosity statistics
                      --optimizations copy-propagation,dead-code fib.out
```

Figure 4.1: Building and testing the project, then compiling and interpreting a program

4.2 Correctness

Details about the tests and their implementation are discussed in Section 2.6 and Section 3.5 respectively. The goal of Unit and Differential Tests is to give some confidence in the correctness of the basic data-structures and of the interpreter on simple inputs. Fuzz Tests, on the other hand, test the code in rare and unpredictable situations, which are most likely to be overlooked when developing code and manually creating tests. Fuzz Tests were highly effective in DLANG-VM, as they found *four* bugs that had not been caught by other tests. Table 4.1 shows these bugs, which are of a relatively hard to predict and hard to test nature.

Wrong version	Corrected version	Description
`jit_movr(cp, dest - 1)`	`jit_movi(cp, dest - 1)`	JIT instruction in a rare branch
`jit_subi(ra, 1, b)`	`jit_movi(ra, 1 - b)`	JIT instruction in a rare branch
`std::stol(token)`	`std::stoi(token)`	Input token integer overflow
`a.ptr == b.ptr`	`a.ptr is b.ptr`	Python reference equality with ==

Table 4.1: The bugs found with Fuzz Tests

4.3 Performance

This Section describes and evaluates the results of performance measurements, which show how the project can be a useful tool for insightful experiments.

Choosing input programs is a very important and often overlooked aspect of benchmarking. I followed the guidelines set out by Partain's "nofib" paper about benchmarking Haskell interpreters [36]. The paper proposes some Haskell programs and splits them into three sets, *Real* (realistic and useful), *Imaginary* (small and pedagogical) and *Spectral* (in between the two), and suggests that benchmarks should prefer Real over Imaginary programs. The suggested Real programs are very large and transforming them from Haskell to DLANG is not feasible in a reasonable time: I therefore use four Imaginary and two Spectral translated programs (listed in Table 4.2), as the former can be quickly implemented, while the latter are more similar to Real programs and can give useful insights on how the interpreter performs on realistic inputs.

Test	Description	Implementation	Classification
`fib`	The Fibonacci function applied to 36	Recursion	Imaginary
`ack`	The Ackermann function applied to (3,9)	Recursion	Imaginary
`prim`	Counts the primes smaller than 40,000	Imperative	Imaginary
`queen`	Places 18 queens on an 18x18 chessboard	Continuations	Imaginary
`sort`	Quicksort of 200 integers, 2000 times	Linked lists	Spectral
`hanoi`	Solves Hanoi Towers with 4 disks	Iterative Deepening	Spectral

Table 4.2: The benchmark inputs and their classification

4.3.1 Just-In-Time Compilation techniques

Figure 4.2 shows the execution times of the input programs on six interpreters: DLANG-VM without JIT compilation, DLANG-VM with Function JIT, its optimised version, DLANG-VM with Tracing JIT, its optimised version and Meta-DLANG-VM (Meta-Tracing JIT). The exact timings are in Appendix E and the choice of optimisations is discussed in Section 4.3.2. Some observations can be made on the results:

- *All forms of JIT compilation are clearly superior to simple interpretation in DLANG-VM.* This is unsurprising, as JIT compilation greatly reduces the time wasted in the fetch and decode steps of the fetch-decode-execute loop.
- *Function and Tracing JIT compilation perform similarly*, but marginally benefit from different program characteristics and paradigms. For example, Function JIT is faster in `fib`, which is implemented by a single function, while Tracing JIT is faster in `ack`, which is implemented by multiple functions.

- *Optimisations reduce execution time in all tests*, with varying effects in Function and Tracing JIT due to their differences in optimisations (also evaluated in Section 4.3.2).
- *The Meta-Tracing JIT interpreter is surprisingly fast*, and even outperforms DLANG-VM (a ten-times-larger C++ program) in `prim` and `sort`. It is however significantly slower in `ack`, `queen` and `hanoi`. By analysing this further with `perf`, `memcpy` operations appear as the main overheads, suggesting that memory management in RPython is the bottleneck.

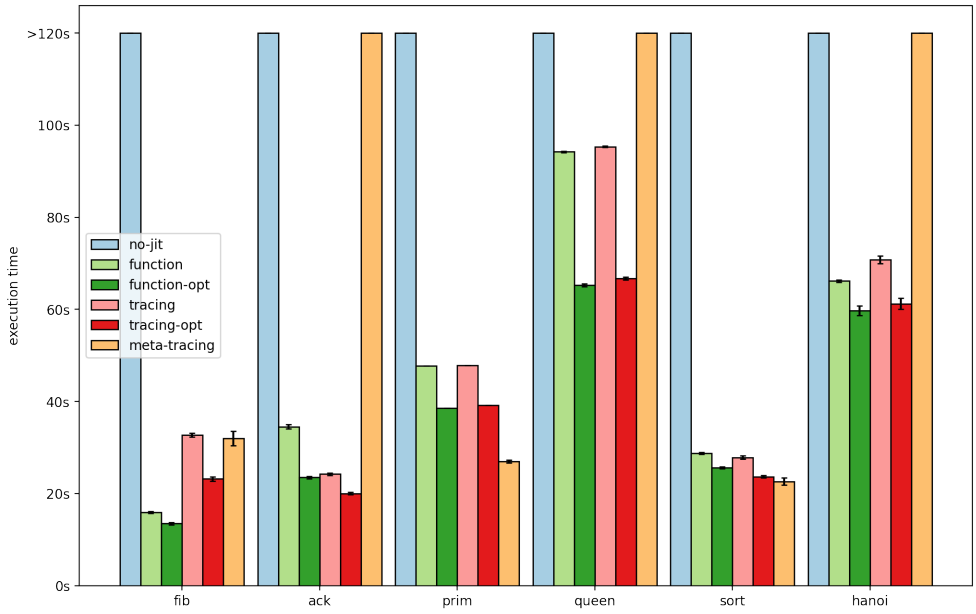


Figure 4.2: Execution times of different interpreters with 95% confidence intervals

4.3.2 Just-In-Time Compiler Optimisations

The previous Section shows that JIT compiler optimisations in DLANG-VM are effective in improving performance, with reductions in execution time ranging from 10% to 30%. This Section analyses in greater detail which ordering of the four implemented optimisations (Section 3.3.2) is most effective. This is called the *Phase-Ordering Problem*, and is an important but in general undecidable [37] problem in the design of optimising compilers.

Figure 4.3 shows the number of instructions in the code after each optimisation, for some sequences of optimisations (computed as the sum over all JIT compiled functions or traces), and the detailed results are in Appendix E. S1 is (empirically) the best optimisations sequence without repetitions on the benchmarks, and the other plots are useful to draw other conclusions.

- *Optimisations have different effects on Function and Tracing JIT* because these have different advantages: with the former, stronger assumptions can be made (for example, that all data on the stack but the return value will be discarded), while the latter enables optimisations spanning multiple function calls.

- *Redundant Checks is the most effective optimisation*, wherever it is placed in the sequence. This suggests that, as expected, run-time checks are significant overheads for dynamically typed languages such as DLANG.
- S1, S2, S3 have Redundant Checks in different positions, and S1 is the best among them.
- S4 swaps the last two optimisations of S1, achieving the same number of instructions.
- S5 adds an additional Redundant Checks at the end. This slightly decreases the number of instructions, because the other optimisations created new redundancies. It follows that *applying optimisations repeatedly is beneficial*, but has an overhead in optimisation time.
- S6 removes Copy Propagation from S3, and is significantly worse than all others: this shows that *Copy Propagation, although not removing any instruction, greatly improves the benefits of Constant Folding, Dead Code Elimination and Redundant Checks.*

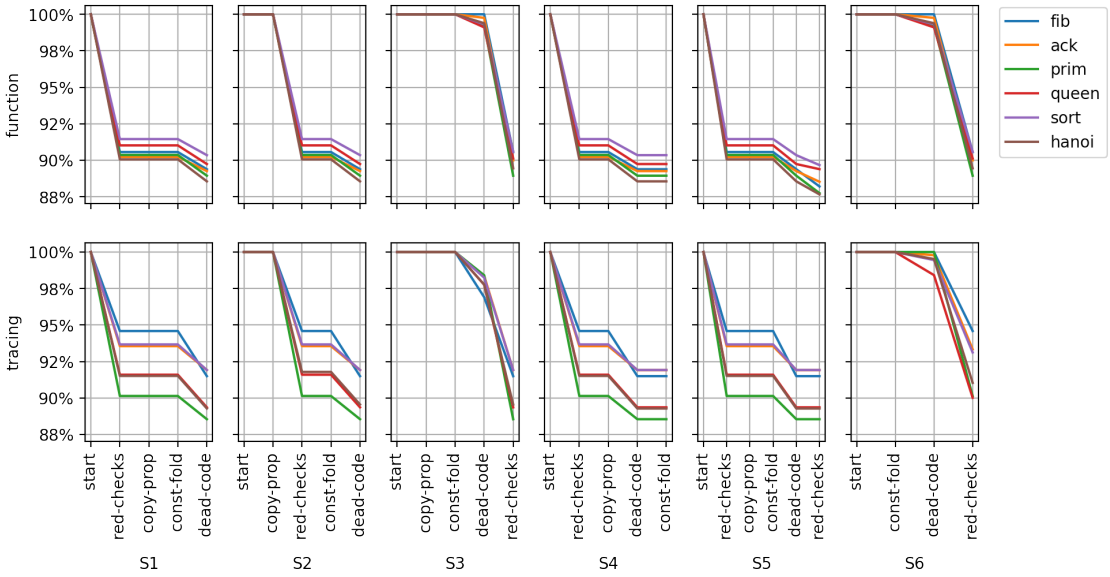


Figure 4.3: Percentage of instructions left after each optimisation step

4.4 Summary

The project successfully passes 21 Unit Tests, 102 Differential Tests, and 15,000 Fuzz Tests, with the last two being applied to a total of 271 interpreters. The code implements complex techniques used in modern interpreters, meets industrial-level stylistic requirements, and its design follows the guidelines of modern C++ and Python, making it more approachable to users interested in the project.

The experiments prove the effectiveness of Just-In-Time compilation, which greatly outperforms simple interpretation. Function JIT and Tracing JIT proved to have different efficacy in different programs, while Meta-Tracing JIT often achieves similar or sometimes better performance, but can be extremely slow in some circumstances.

Chapter 5

Conclusion

5.1 Project Outcome

The project successfully achieves all success criteria: DLANG-C, DLANG-VM with Function JIT and correctness tests. It also achieves additional goals, including:

- Tracing JIT, optimisations and memory management for DLANG-VM, making it a performant and versatile tool with most components of real interpreters on a smaller scale.
- Meta-DLANG-VM, a short and simple RPython script that provides all functionalities of the much larger DLANG-VM and achieves similar performances;
- additional tests validating the project and facilitating any future extensions, including thousands of fully automated randomised tests.

The project also brings further proof of how powerful some of the tools being studied in academia and used by the best interpreters are, while making them more accessible:

- **Just-In-Time compilation**: many modern interpreted languages would be orders of magnitude slower without JIT compilation, and the same is proved to be true for DLANG.
- **Meta-Tracing Just-In-Time compilation**: even though Meta-Tracing interpreters are in general outperformed by ad-hoc interpreters, especially if heavily optimised by thousands of developers over decades, this project proves that for small interpreters developed with scarce resources, it is an exceptionally cost-effective and surprisingly performant solution.
- **Fuzz Testing**: Fuzzing can generate inputs that humans would struggle to create, and for compilers and interpreters it is even easier to set up and more effective, as it can take advantage of the strict Context Free Grammar input structure.

5.2 Future Work

Modularity is one of the core design principles of DLANG-VM. Therefore, there are many new modular extensions that could be implemented in the future using DLANG-VM's extensions API, in particular for other optimisations and memory managers: for both, many algorithms exist, and implementing them on DLANG-VM can be a great way to learn and test them. In particular, *Register Allocation* is a useful optimisation not currently implemented.

As far as work on more fundamental aspects of DLANG-VM is concerned, there are many interesting ideas inspired by existing interpreters. One is multi-threading: modern JIT compiling interpreters do not generally stop interpretation to JIT compile code, but do so on a separate thread, improving performance and more specifically responsiveness. Other possible extensions are *Parallel Garbage Collectors*, which use multiple threads to improve the performance of collection, and *Concurrent Garbage Collectors*, which run concurrently with the program.

5.3 Lessons Learnt

Several aspects of the project have been a source of technical growth and improvement for me. First and foremost, it was proof of the importance of carefully planning the multiple months of work at the very start, but with the final goals in mind. Milestones, clearly defined deliverables and awareness of their dependencies make progress more consistent, making it easier to understand when parts of the project are not going to plan, and to adapt in a timely manner.

Prioritising testing has also been extremely beneficial: when code is well tested, new features can be implemented with confidence in the older features they interact with, simplifying error detection. In addition to that, following design and stylistic guidelines made my code easier to read and understand, contributing to a much smoother development experience.

Balancing new ideas with existing algorithms in the literature and new implementations with existing tools and libraries, are trade-offs I often had to deal with while working on the project. During its initial stages, I sometimes found myself "re-inventing the wheel", and finding the right balance is a useful skill I developed over time.

Finally, the timeline set out at the start was probably over-pessimistic, as I expected some parts of the project to require more time than they actually did, and this resulted in the last additions lacking the bigger-picture planning of other components. Finding ways to extend the project in a cohesive way at the start is much easier than at the end, and this is another useful lesson.

Bibliography

[1] John W Backus, Robert J Beeber, Sheldon Best, Richard Goldberg, Lois M Haibt, Harlan L Herrick, Robert A Nelson, David Sayre, Peter B Sheridan, Harold Stern, et al. The fortran automatic coding system. In *Papers presented at the February 26-28, 1957, western joint computer conference: Techniques for reliability*, pages 188–198, 1957.

[2] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, 1960.

[3] Paul Klint. Interpretation techniques. *Software: Practice and Experience*, 11(9):963–973, 1981.

[4] Stephen O'Grady. The redmonk programming language rankings: June 2021, Aug 2021. Available at `https://redmonk.com/sogrady/2021/08/05/language-rankings-6-21/`.

[5] Timothy Cramer, Richard Friedman, Terrence Miller, David Seberger, Robert Wilson, and Mario Wolczko. Compiling java just in time. *Ieee micro*, 17(3):36–43, 1997.

[6] Timothy G Griffin. Slang. Available at `https://github.com/Timothy-G-Griffin/cc_cl_cam_ac_uk/tree/master/slang`.

[7] L Peter Deutsch and Allan M Schiffman. Efficient implementation of the smalltalk-80 system. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 297–302, 1984.

[8] Andreas Gal, Christian W Probst, and Michael Franz. Hotpathvm: An effective jit compiler for resource-constrained devices. In *Proceedings of the 2nd international conference on Virtual execution environments*, pages 144–153, 2006.

[9] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, et al. Trace-based just-in-time type specialization for dynamic languages. *ACM Sigplan Notices*, 44(6):465–478, 2009.

[10] Carl Friedrich Bolz and Laurence Tratt. The impact of meta-tracing on vm design and implementation. *Science of Computer Programming*, 98:408–421, 2015.

[11] John Cocke. Global common subexpression elimination. In *Proceedings of a symposium on Compiler optimization*, pages 20–24, 1970.

[12] Ken Kennedy. A global flow analysis algorithm. *International Journal of Computer Mathematics*, 3(1-4):5–15, 1972.

[13] Chris J Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, 1970.

[14] Maarten Vandercammen, Stefan Marr, and Coen De Roover. A flexible framework for studying trace-based just-in-time compilation. *Computer Languages, Systems & Structures*, 51:22–47, 2018.

[15] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. Tracing the meta-level: Pypy's tracing jit compiler. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 18–25, 2009.

[16] Gnu lightning manual. Available at `https://www.gnu.org/software/lightning/manual/lightning.html`.

[17] Jason Merrill. Generic and gimple: A new tree representation for entire functions. In *Proceedings of the 2003 GCC Developers' Summit*, pages 171–179. Citeseer, 2003.

[18] David F Bacon, Perry Cheng, and VT Rajan. A unified theory of garbage collection. In *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 50–68, 2004.

[19] Gregory Colvin. Exception safe smart pointers. *C++ committee document*, pages 94–168, 1994.

[20] Ken Thompson. Reflections on trusting trust. *Communications of the ACM*, 27(8):761–763, 1984.

[21] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[22] William M McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

[23] Flash Sheridan. Practical testing of a c99 compiler using output comparison. *Software: Practice and Experience*, 37(14):1475–1488, 2007.

[24] Joe W Duran and Simeon C Ntafos. An evaluation of random testing. *IEEE transactions on Software Engineering*, SE-10(4):438–444, 1984.

[25] Mariano Ceccato, Alessandro Marchetto, Leonardo Mariani, Cu D Nguyen, and Paolo Tonella. An empirical study about the effectiveness of debugging when random test cases are used. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 452–462. IEEE, 2012.

[26] Google. Oss-fuzz. Available at `https://google.github.io/oss-fuzz/`.

[27] Donald G Malcolm, John H Roseboom, Charles E Clark, and Willard Fazar. Application of a technique for research and development program evaluation. *Operations research*, 7(5):646–669, 1959.

[28] Henry Laurence Gantt. *Work, Wages, and Profits*. Industrial management library. Engineering Magazine Company, 1913.

[29] Barry W. Boehm. A spiral model of software development and enhancement. *Computer*, 21(5):61–72, 1988.

[30] Lawrence Crowl and Matt Austern. C++ dynamic arrays. *ISO/IEC JTC1 SC22 WG21 N3662*, 2013.

[31] Rpython's documentation. Available at `https://rpython.readthedocs.io/en/latest/index.html`.

[32] Mozilla Security. Dharma. Available at `https://github.com/posidron/dharma`.

[33] Google. Google c++ style guide. Available at `https://google.github.io/styleguide/cppguide.html`.

[34] Guido van Rossum. Pep 8 – style guide for python code. Available at `https://peps.python.org/pep-0008/`.

[35] OCaml. Ocaml programming guidelines. Available at `https://ocaml.org/learn/tutorials/guidelines.html`.

[36] Will Partain. The nofib benchmark suite of haskell programs. In *Functional Programming, Glasgow 1992*, pages 195–202. Springer, 1993.

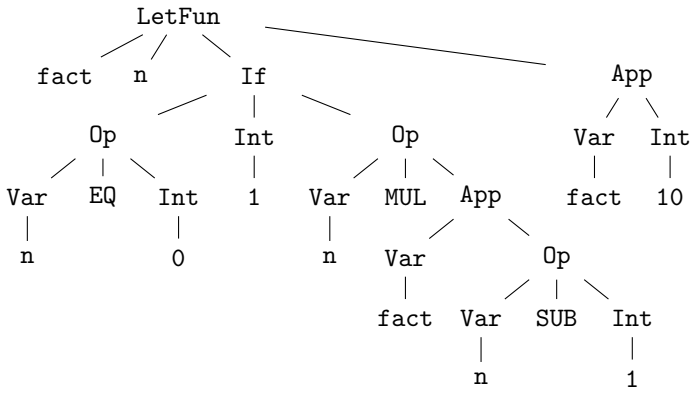
[37] Sid-Ahmed-Ali Touati and Denis Barthou. On the decidability of phase ordering problem in optimizing compilation. In *Proceedings of the 3rd Conference on Computing Frontiers*, pages 147–156, 2006.

Appendix A

Example of Code Transformations

The input to DLANG-C is a high-level DLANG program, in this example computing 10!. The input is tokenised and then converted to the Parsed AST. The Parsed AST is then transformed into the AST after applying static checks and removing syntactic sugar. In this small example, the two trees are identical.

```
1 let fact n =
2   if n = 0 then
3     1
4   else
5     n * (fact (n - 1))
6   end
7 in
8   fact 10
9 end
```



The AST is then transformed by DLANG-C into byte-code. Here, label `L0` is the implicit function implementing line 8 of the program, while label `L1` implements the `fact` function. Labels `L2` and `L3` are the destination of lines 14 and 16, and implement the `if` statement.

```
1  MK_CLOSURE L1 0
2  MK_CLOSURE L0 0
3  APPLY
4  HALT
5  FUNCTION L0
6  PUSH STACK_INT 10
7  LOOKUP STACK_LOC -2
8  APPLY
9  RETURN
```

```
10 FUNCTION L1
11 LOOKUP STACK_LOC -2
12 PUSH STACK_INT 0
13 OPER EQ
14 TEST L2
15 PUSH STACK_INT 1
16 GOTO L3
```

```
17 LABEL L2
18 LOOKUP STACK_LOC -2
19 LOOKUP STACK_LOC -2
20 PUSH STACK_INT 1
21 OPER SUB
22 LOOKUP STACK_LOC -1
23 APPLY
24 OPER MUL
25 LABEL L3
26 RETURN
```

DLANG-VM interprets the byte-code and, in the case of Tracing JIT, compiles the trace `10 11 12 13 14 17 18 19 20 21 22 23`, the `fact` function in the `n > 0` case. The trace is first converted to U-DLANG, then to T-DLANG, optimised, and then back to U-DLANG to be JIT compiled. These are the transformations (for brevity, 45 out of 122 instructions are included):

```
LABEL 10                      LABEL                 LABEL                 LABEL 10
OPER cp cp + 1                OPER x1 x1 + imm      OPER x1 x1 + imm      OPER cp cp + 1
LABEL 11                      LABEL                 LABEL                 LABEL 11
MEM-CHECK (fp-2)->tag         MEM-CHECK x3          MEM-CHECK x3          MEM-CHECK (fp-2)->tag
GET r0 (fp-2)->tag            MOVE x4 x3            MOVE x4 x3            GET r0 (fp-2)->tag
MEM-CHECK (sp+0)->tag         MEM-CHECK x6          MEM-CHECK x6          MEM-CHECK (sp+0)->tag
SET (sp+0)->tag r0            MOVE x6 x4            MOVE x6 x4            SET (sp+0)->tag r0
MEM-CHECK (fp-2)->val         MEM-CHECK x7          MEM-CHECK x7          MEM-CHECK (fp-2)->val
GET r0 (fp-2)->val            MOVE x4 x7            MOVE x4 x7            GET r0 (fp-2)->val
MEM-CHECK (sp+0)->val         MEM-CHECK x8          MEM-CHECK x8          MEM-CHECK (sp+0)->val
SET (sp+0)->val r0            MOVE x8 x4            MOVE x8 x4            SET (sp+0)->val r0
OPER sp sp + 1                OPER x5 x5 + imm      OPER x5 x5 + imm      OPER sp sp + 1
OPER cp cp + 1                OPER x1 x1 + imm      OPER x1 x1 + imm      OPER cp cp + 1
LABEL 12                      LABEL                 LABEL                 LABEL 12
MEM-CHECK (sp+0)->tag         MEM-CHECK x9          MEM-CHECK x9          MEM-CHECK (sp+0)->tag
SET (sp+0)->tag Int           MOVE x9 imm           MOVE x9 imm           SET (sp+0)->tag Int
MEM-CHECK (sp+0)->val         MEM-CHECK x10         MEM-CHECK x10         MEM-CHECK (sp+0)->val
SET (sp+0)->val 0             MOVE x10 imm          MOVE x10 imm          SET (sp+0)->val 0
OPER sp sp + 1                OPER x5 x5 + imm      OPER x5 x5 + imm      OPER sp sp + 1
OPER cp cp + 1                OPER x1 x1 + imm      OPER x1 x1 + imm      OPER cp cp + 1
LABEL 13                      LABEL                 LABEL                 LABEL 13
MEM-CHECK (sp-2)->tag         MEM-CHECK x6          MEM-CHECK x6          MEM-CHECK (sp-2)->tag
GET r0 (sp-2)->tag            MOVE x4 x6            MOVE x4 x6            GET r0 (sp-2)->tag
MEM-CHECK (sp-1)->tag         MEM-CHECK x9          MEM-CHECK x9          MEM-CHECK (sp-1)->tag
GET r1 (sp-1)->tag            MOVE x11 x9
OPER r0 r0 = r1               OPER x4 x4 = x11      OPER x4 x4 = imm      OPER r0 r0 = Int
MEM-CHECK (sp-2)->val         MEM-CHECK x8          MEM-CHECK x8          MEM-CHECK (sp-2)->val
GET r1 (sp-2)->val            MOVE x11 x8           MOVE x11 x8           GET r1 (sp-2)->val
MEM-CHECK (sp-1)->val         MEM-CHECK x10         MEM-CHECK x10         MEM-CHECK (sp-1)->val
GET r2 (sp-1)->val            MOVE x12 x10
OPER r1 r1 = r2               OPER x11 x11 = x12    OPER x11 x11 = imm    OPER r1 r1 = 0
OPER r0 r0 & r1               OPER x4 x4 & x11      OPER x4 x4 & x11      OPER r0 r0 & r1
MEM-CHECK (sp-2)->val         MEM-CHECK x8
SET (sp-2)->val r0            MOVE x8 x4            MOVE x8 x4            SET (sp-2)->val r0
MEM-CHECK (sp-2)->tag         MEM-CHECK x6
SET (sp-2)->tag Bool          MOVE x6 imm           MOVE x6 imm           SET (sp-2)->tag Bool
OPER sp sp - 1                OPER x5 x5 - imm      OPER x5 x5 - imm      OPER sp sp - 1
OPER cp cp + 1                OPER x1 x1 + imm      OPER x1 x1 + imm      OPER cp cp + 1
LABEL 14                      LABEL                 LABEL                 LABEL 14
MEM-CHECK (sp-1)->val         MEM-CHECK x8          MEM-CHECK x8          MEM-CHECK (sp-1)->val
TAG-CHECK Bool (sp-1)->tag    TAG-CHECK Bool x6     TAG-CHECK Bool imm    TAG-CHECK Bool Bool
GET r0 (sp-1)->val            MOVE x4 x8            MOVE x4 x4            MOVE r0 r0
OPER sp sp - 1                OPER x5 x5 - imm      OPER x5 x5 - imm      OPER sp sp - 1
BRANCH 16 r0                  BRANCH x4             BRANCH x4             BRANCH 16 r0
GUARD                         GUARD                 GUARD                 GUARD
...                           ...                   ...                   ...
```

Some noteworthy details are:

- U-DLANG uses registers and memory locations, while T-DLANG only uses variables;
- optimisations successfully removed 4 instructions;
- a `GUARD` statement is added at the end to check if the code is following the expected path.

Appendix B

DLANG CFG, Tags and Byte-Code

The basic data types are `unit`, `bool` and `int`, which can be combined into composite data types as: `A -> B`, `A * B`, `A + B`, `A ref`. The *Context Free Grammar* of the language is defined as follows (statements using type hints for compatibility with SLANG are omitted):

```
<Int> := ...

<Var> := ...

<SimpleExpr> :=
  ()
  true
  false
  <Int>
  <Var>
  ?
  ( <Expr> )
  ( <Expr> , <Expr> )
  ~ <SimpleExpr>
  !  <SimpleExpr>
  ref <SimpleExpr>
```

```
<Expr> :=
  <SimpleExpr>
  <Expr> <SimpleExpr>
  - <SimpleExpr>
  <Expr> < <Expr>
  <Expr> - <Expr>
  <Expr> * <Expr>
  <Expr> / <Expr>
  <Expr> < <Expr>
  <Expr> = <Expr>
  <Expr> && <Expr>
  <Expr> || <Expr>
  <Expr> := <Expr>
```

```
  begin <Exprlist> end
  if <Expr> then <Expr> else <Expr> end
  while <Expr> do <Expr> end
  fst <Expr>
  snd <Expr>
  fun <Var> -> <Expr> end
  let <Var> = <Expr> in <Expr> end
  let <Var> <Var> = <Expr> in <Expr> end
  inl <Expr>
  inr <Expr>
  case <Expr> of inl <Var> -> <Expr>
                | inr <Var> -> <Expr> end

<Exprlist> :=
  <Expr>
  <Expr> ; <Exprlist>
```

The tags on the stack are:

- `Unit`, `Bool`, `Int`: simple values;
- `HeapIndex` `HeapRef`: pointers to the heap for complex objects and references respectively;
- `CodeIndex`, `ReturnAddress`, `FramePointer`: components of stack frames and closures.

The tags on the heap are those on the stack with the addition of:

- `PairHeader`: precedes two pair elements;
- `InlHeader`, `InrHeader`: precedes an element of a union with options "left" and "right";
- `ClosureHeader`: precedes a list of values for variables in the closure.

DLANG Byte-Code instructions are defined as follows:

Byte-Code Instruction	Description
`UNARY OP`	Apply `OP` ($\neg$, $-$, read) on the item on the top of the stack
`OPER OP`	Apply `OP` ($\vee$, $\wedge$, $=$, $<$, $+$, $-$, $\times$, $\div$) on the two items on the top of the stack
`MK_PAIR`	Make a pair from the two items on top of the stack
`FST`	Get the first element of the pair on the top of the stack
`SND`	Get the second element of the pair on the top of the stack
`MK_INL`	Make a case "left" object from the item on the top of the stack
`MK_INR`	Make a case "right" object from the item on the top of the stack
`PUSH T X`	Push `X` of type `T` (unit, bool, int) on the top of the stack
`APPLY`	Apply the closure on the top of the stack to the argument below
`RETURN`	Remove the stack frame, preserving the item on the top, and restore `cp`, `fp`, `sp`
`LOOKUP P K`	Copy the item with offset `K` from `P` (frame pointer or heap base)
`MK_CLOS N`	Make a closure with the `N` elements on the top of the stack
`SWAP`	Swap the two elements on the top of the stack
`POP`	Remove the element on the top of the stack
`LABEL L`	Mark the start of a sequence of instructions with label `L`
`FUNCTION L`	Mark the start of a function body with label `L`
`DEREF`	Get the value of the reference on the top of the stack
`MK_REF`	Make a reference from the value on the top of the stack
`ASSIGN`	Assign the value on the top of the stack to the reference below
`HALT`	Halt the execution of the program
`GOTO L`	Go to label `L`
`TEST L`	Go to label `L` if `True` is on the top of the stack
`CASE L`	Go to label `L` if a case "right" is on the top of the stack

Appendix C

Command-Line Arguments

This is the help message of DLANG-VM:

```
Usage: dlang_vm [options] [<file>]:
  --help                    Display this list of options
  --verbosity arg (=output) One of:
                            - quiet:      no output is produced
                            - output:     program output to stdout
                            - time:       timing statistics to stderr
                            - statistics: all statistics to stderr
                            - debug:      output to stdout, debug to stderr
  --jit-threshold arg (=0)  The value of the threshold for jit compilation
  --jit-policy arg (=no)    One of:
                            - no
                            - tracing
                            - individual (group jit)
                            - block (group jit)
                            - function (group jit)
  --memory arg (=none)      One of:
                            - none
                            - amortized
                            - mark-and-sweep
  --optimizations arg       A comma-separated list of:
                            - redundant-checks
                            - unused-writes
                            - dead-code
                            - copy-propagation
```

These are the help messages of the scripts for building and testing:

```
Usage:
  ./build.py clean
  ./build.py slang
  ./build.py dlang-c
  ./build.py dlang-vm
  ./build.py meta
  ./build.py all
```

```
Usage:
  ./test.py differential
  ./test.py unit
  ./test.py fuzz
```

Appendix D

Proof of Confidence Interval

I assume that the samples of the time duration of a program are i.i.d. random variables of a Normal Distribution with mean μ and standard deviation σ. A 95% confidence interval is:

$$\left[\mu - z_{0.025} \cdot \sigma \cdot \sqrt{n}, \mu + z_{0.025} \cdot \sigma \cdot \sqrt{n}\right] \qquad z_{0.0025} \approx 1.96$$

Since the values of μ and σ are unknown, these must be estimated. Let the n samples be $X_1, X_2, X_3, \ldots, X_n \sim N(\mu, \sigma)$. Let the *Sample Mean* $\overline{X}_n$ and *Sample Variance* S_n^2 be:

$$\overline{X}_n = \frac{1}{n} \cdot \sum_{i=1}^{n} X_i \qquad S_n^2 = \frac{1}{n-1} \cdot \sum_{i=1}^{n} \left(X_i - \overline{X}_n\right)^2$$

From the properties of expectation and variance, it follows that:

$$\begin{aligned}
E\left[\overline{X}_n\right] &= \frac{1}{n} \cdot E\left[\sum_{i=1}^{n} X_i\right] = \frac{1}{n} \cdot \sum_{i=1}^{n} E\left[X_i\right] = \frac{1}{n} \cdot \sum_{i=1}^{n} \mu = \mu \\
Var\left[\overline{X}_n\right] &= \frac{1}{n^2} \sum_{i=1}^{n} Var\left[\overline{X}_i\right] = \frac{1}{n^2} \sum_{i=1}^{n} \sigma^2 = \frac{\sigma^2}{n} \\
E\left[S_n^2\right] &= \frac{1}{n-1} \cdot E\left[\sum_{i=1}^{n} \left(X_i^2 - 2 \cdot X_i \cdot \overline{X}_n + \overline{X}_n^2\right)\right] \\
&= \frac{1}{n-1} \cdot \left(\sum_{i=1}^{n} E\left[X_i^2\right] - 2 \cdot E\left[\overline{X}_n \cdot \sum_{i=1}^{n} X_i\right] + \sum_{i=1}^{n} E\left[\overline{X}_n^2\right]\right) \\
&= \frac{1}{n-1} \cdot \left(\sum_{i=1}^{n} E\left[X_i^2\right] - 2 \cdot E\left[\overline{X}_n \cdot \left(n \cdot \overline{X}_n\right)\right] + \sum_{i=1}^{n} E\left[\overline{X}_n^2\right]\right) \\
&= \frac{1}{n-1} \cdot \left(n \cdot \left(\sigma^2 + \mu^2\right) - 2 \cdot n \cdot \left(\frac{\sigma^2}{n} + \mu^2\right) + n \cdot \left(\frac{\sigma^2}{n} + \mu^2\right)\right) = \sigma^2
\end{aligned}$$

This means that the 95% confidence interval can be computed from the samples as follows:

$$\left[\overline{X}_n - z_{0.025} \cdot S_n \cdot \sqrt{n}, \overline{X}_n + z_{0.025} \cdot S_n \cdot \sqrt{n}\right] \qquad z_{0.0025} \approx 1.96$$

Appendix E

Experimental Results

These are the results of experiments in Section 4.3.1, obtained from 30 samples for each test.

	no-jit	function	function-opt	tracing	tracing-opt	meta-tracing
fib	> 120000 ms	15906 ± 146 ms	13478 ± 140 ms	32707 ± 255 ms	23193 ± 273 ms	32001 ± 935 ms
ack	> 120000 ms	34521 ± 283 ms	23485 ± 145 ms	24208 ± 174 ms	20024 ± 158 ms	> 120000 ms
prim	> 120000 ms	47715 ± 16 ms	38531 ± 21 ms	47806 ± 15 ms	39160 ± 14 ms	26953 ± 186 ms
queen	> 120000 ms	94196 ± 98 ms	65221 ± 185 ms	95310 ± 86 ms	66714 ± 177 ms	> 120000 ms
sort	> 120000 ms	28719 ± 110 ms	25633 ± 130 ms	27823 ± 212 ms	23655 ± 147 ms	22624 ± 481 ms
hanoi	> 120000 ms	66168 ± 165 ms	59670 ± 621 ms	70784 ± 501 ms	61219 ± 706 ms	> 120000 ms

These are the results of experiments in Section 4.3.2. Each cell represents the sum of the number of instructions in all JIT compiled functions or traces. The following abbreviations are used for optimisations:

- `checks`: Redundant Checks
- `copy`: Copy Propagation
- `const`:Constant Folding
- `dead`: Dead Code Elimination

		start	checks	copy	const	dead
function	fib	339	307	307	307	303
	ack	819	739	739	739	731
	prim	1338	1209	1209	1209	1190
	queen	3128	2847	2847	2847	2807
	sort	3087	2823	2823	2823	2789
	hanoi	3913	3524	3524	3524	3465
tracing	fib	388	367	367	367	355
	ack	915	856	856	856	841
	prim	1257	1133	1133	1133	1113
	queen	6887	6308	6308	6308	6154
	sort	3822	3580	3580	3580	3513
	hanoi	11723	10728	10728	10728	10466

		start	copy	checks	const	dead
function	fib	339	339	307	307	303
	ack	819	819	739	739	731
	prim	1338	1338	1209	1209	1190
	queen	3128	3128	2847	2847	2807
	sort	3087	3087	2823	2823	2789
	hanoi	3913	3913	3524	3524	3465
tracing	fib	388	388	367	367	355
	ack	915	915	856	856	841
	prim	1257	1257	1133	1133	1113
	queen	6887	6887	6308	6308	6154
	sort	3822	3822	3580	3580	3513
	hanoi	11723	11723	10760	10760	10498

		start	copy	const	dead	checks
function	fib	339	339	339	339	307
	ack	819	819	819	817	737
	prim	1338	1338	1338	1328	1190
	queen	3128	3128	3128	3100	2819
	sort	3087	3087	3087	3066	2796
	hanoi	3913	3913	3913	3889	3500
tracing	fib	388	388	388	376	355
	ack	915	915	915	900	841
	prim	1257	1257	1257	1237	1113
	queen	6887	6887	6887	6733	6154
	sort	3822	3822	3822	3755	3513
	hanoi	11723	11723	11723	11461	10498

		start	checks	copy	dead	const
function	fib	339	307	307	303	303
	ack	819	739	739	731	731
	prim	1338	1209	1209	1190	1190
	queen	3128	2847	2847	2807	2807
	sort	3087	2823	2823	2789	2789
	hanoi	3913	3524	3524	3465	3465
tracing	fib	388	367	367	355	355
	ack	915	856	856	841	841
	prim	1257	1133	1133	1113	1113
	queen	6887	6308	6308	6154	6154
	sort	3822	3580	3580	3513	3513
	hanoi	11723	10728	10728	10466	10466

		start	checks	copy	const	dead	checks
function	fib	339	307	307	307	303	299
	ack	819	739	739	739	731	725
	prim	1338	1209	1209	1209	1190	1174
	queen	3128	2847	2847	2847	2807	2796
	sort	3087	2823	2823	2823	2789	2768
	hanoi	3913	3524	3524	3524	3465	3430
tracing	fib	388	367	367	367	355	355
	ack	915	856	856	856	841	841
	prim	1257	1133	1133	1133	1113	1113
	queen	6887	6308	6308	6308	6154	6154
	sort	3822	3580	3580	3580	3513	3513
	hanoi	11723	10728	10728	10728	10466	10466

		start	const	dead	checks
function	fib	339	339	339	307
	ack	819	819	817	737
	prim	1338	1338	1328	1190
	queen	3128	3128	3100	2819
	sort	3087	3087	3066	2796
	hanoi	3913	3913	3889	3500
tracing	fib	388	388	388	367
	ack	915	915	913	854
	prim	1257	1257	1257	1133
	queen	6887	6887	6778	6199
	sort	3822	3822	3801	3559
	hanoi	11723	11723	11667	10672

Project Proposal

Project introduction

The project involves the implementation of some program execution tools for DLANG, a minimalistic language designed specifically for this project as a dynamically typed version of the SLANG language from the Compiler Construction course. First of all, I shall implement a compiler from DLANG source code to DLANG byte-code. Secondly, I shall implement DLANG-VM, a Virtual Machine for the execution of DLANG, whose core component is a stack-oriented interpreter of DLANG byte-code using Just-In-Time Compilation. Just-In-Time Compilation is a hybrid between Ahead-Of-Time Compilation and Interpretation, which has proven to be successful in many industrial-strength Virtual Machines, such as the JVM.

Description of the work

The initial part of the project is the implementation of a compiler of DLANG source code to byte-code. I will not create it from scratch, but starting from the SLANG compiler implemented in OCaml as part of the Compiler Construction course. Some modifications to SLANG and its compiler are required to make it a dynamically typed language: type declarations in the source code are ignored, static type checking is removed and in the internal syntax trees all typed instructions are substituted by non-typed instructions (e.g. both EQI and EQB become EQ). A benefit of these design decisions is that, as far as source code syntax is concerned, SLANG and DLANG are identical languages. In addition to that, all well-typed SLANG programs are valid DLANG programs, but there exist some DLANG programs that execute without errors on DLANG-VM while giving compilation errors on the SLANG compiler. I shall implement DLANG-VM, the main body of the project, using C++ as the programming language and CMake as the build tool. The core of DLANG-VM is the byte-code interpreter, which interprets and executes byte-code at run-time. DLANG-VM's interpreter will provide Just-In-Time Compilation, which works as follows:

- The usage frequency of functions is tracked during regular interpretation.
- When a certain usage threshold is exceeded, the function is marked as "frequent" and compiled to x86 machine-code.
- Whenever a "frequent" function is called again and its arguments have the expected types, it is not interpreted, but its pre-compiled native code is executed.

More advanced versions of Just-In-Time compilation are optional extensions for the project, but not part of the success criteria. To make the testing phase simpler and the test results more useful, DLANG-VM will provide the option to activate or deactivate JIT compilation and some parameters to fine-tune the policy used to determine which functions are marked as "frequent". In addition to JIT compilation, DLANG-VM will provide basic memory management. In particular, memory is allocated in a contiguous section and never de-allocated, which means that DLANG-VM stops if the maximum memory size is exceeded. Testing for this project has two goals, testing the correctness of DLANG-VM and measuring the performance benefits of Just-In-Time Compilation on DLANG byte-code. As far as correctness is concerned, I shall use the interpreter from the course as the gold standard and compare program outputs. For performance measurements and the evaluation of DLANG-VM, multiple tests can be performed:

- Comparing the performance of programs running on DLANG-VM with activated JIT compilation and programs running with deactivated JIT compilation. This will show whether JIT compilation is useful to achieve some running time improvements.
- Comparing the performance of DLANG-VM with the SLANG compiler to x86 from the course. The theoretical expectation is for the latter to be significantly faster, and the test will show if that is what actually happens in practice.
- Comparing the performance of programs running on DLANG-VM with different parameters for JIT compilation. This will show which parameters have a greater impact on the performance of DLANG-VM and which are the best parameters for the different kinds of programs.

The programs used for testing are the SLANG examples and tests included in the compiler from the course, handwritten programs for more specific tests and automatically generated programs testing the performance of JIT compilation on large programs.

Starting Point

A small (approximately 10% to 20% of the work) but essential part of the project involves the modification of the existing SLANG compiler from the Compiler Construction course to implement the compiler from DLANG source code to DLANG byte-code. This piece of software will therefore be the result of both my work and of the work of Professor Griffin. In addition

to that, some of the example programs included in the SLANG repository will be used to test DLANG-VM, together with other test programs I will create.

Success Criteria

The success criteria for the project are:

- Implementing a compiler from DLANG source code to DLANG byte-code by modifying the existing SLANG compiler from the Compiler Construction course.
- Implementing a Virtual Machine, DLANG-VM, capable of:
  - interpreting DLANG byte-code;
  - performing very basic Just-In-Time compilation;
  - allocating (but not necessarily de-allocating) memory.
- Running tests that successfully establish the correctness of DLANG-VM, using the existing SLANG interpreter as the gold standard to compare program outputs.

Work Plan

Each line represents two weeks of work:

1. October 21st: modifications to the existing SLANG compiler and implementation of the interpreter and memory manager for DLANG-VM.
2. November 4th: implementation of the JIT compiler.
3. November 18th: implementation of the policy determining which code is compiled.
4. December 2nd: testing correctness of DLANG-VM.
5. February 3rd: achievement of all success criteria and production of the Progress Report.
6. February 17th: creation of test programs and test framework for performance testing.
7. March 3rd: execution of performance tests and analysis of results.
8. March 17th: production of the Dissertation and possible extensions.
9. March 31st: production of the Dissertation and possible extensions, full draft Dissertation sent to Supervisor and Directors of Studies.
10. April 14th: dissertation adjustments to reflect Supervisor and Directors of Studies feedback and to incorporate extensions.

Resource declaration

None.