EXAMENSARBETE INOM INFORMATIONSTEKNIK,
AVANCERAD NIVÅ, 30 HP
STOCKHOLM, SVERIGE 2018

A Scala DSL for Rust code generation

KLAS SEGELJAKT

KTH
SKOLAN FÖR ELEKTROTEKNIK OCH DATAVETENSKAP

Abstract

Continuous Deep Analytics (CDA) is a new form of analytics with performance requirements exceeding what the current generation of distributed systems can offer. This thesis is part of a five year project in collaboration between RISE SICS and KTH to develop a next generation distributed system capable of CDA. The two issues which the system aims to solve are *computation sharing* and *hardware acceleration*. The former refers to how BigData and machine learning libraries such as TensorFlow, Pandas and Numpy must collaborate in the most efficient way possible. Hardware acceleration relates to how the back-end of current generation general purpose data processing systems such as Spark and Flink are bottlenecked by the Java Virtual Machine (JVM). As the JVM abstracts over the underlying hardware, its applications become portable but also forfeit the opportunity to fully exploit the available hardware resources. This thesis aims to explore the area of Domain Specific Languages (DSLs) and code generation as a solution to hardware acceleration. The idea is to translate incoming queries to the system into low-level code, tailor suited to each worker machine's specific hardware. To this end, two Scala DSLs for generating Rust code have been developed for the translation step. Rust is a new, low-level programming language with a unique take on memory management which makes it as safe as Java and fast as C. Scala is a language which is well suited towards development of DSLs due to its flexible syntax and semantics. The first DSL is implemented as a string interpolator. The interpolator splices strings of Rust code together, at compile time or runtime, and passes the result to an external process for static checking. The second DSL instead provides an API for constructing an abstract syntax tree, which after construction can be traversed and printed into Rust source code. The API combines three concepts: heterogeneous lists, fluent interfaces, and algebraic data types. These allow the user to express advanced Rust syntax such as polymorphic structs, functions, and traits, without sacrificing type safety.

Keywords— Continuous Deep Analytics, Domain Specific Langauges, Code Generation, Rust, Scala

Sammanfattning

Kontinuerlig Djup Analys (CDA) är en ny form av analys med prestandakrav som överstiger vad den nuvarande generationen av distributerade system kan erbjuda. Den här avhandlingen är del av ett project mellan RISE SICS och KTH för att utveckla ett nästa-generations distribuerat system kapabelt av CDA. Det är två problem som systemet syftar på att lösa: *hårdvaruacceleration* och *beräkningsdelning*. Det första handlar om hur BigData och maskininlärningssystem som sådan som TensorFlow, Pandas och Numpy måste kunna samarbeta så effektivt som möjligt. Hårdvaruacceleration relaterar till hur back-end delen i den dagens distribuerade beräknings system, såsom Spark och Flink, flaskhalsas av Javas Virtuella Maskin. JVM:en abstraherar över den underliggande hårvaran. Som resultat blir dess applikationer portabla, men ger också upp möjligheten att fullständigt utnyttja de tillgängliga hårdvaruresurserna. Den här avhandlingen siktar på att utforska området kring Domänspecifika Språk (DSLer) och kodgenerering som en lösning till hårdvaruacceleration. Idén är att översätta inkommande förfrågningar till låg-nivå kod, skräddarsydd till varje arbetar maskin's specifika hårdvara. Till detta ändamål har två Scala DSLer utvecklats för generering av Rust kod. Rust är ett nytt låg-nivå språk med ett unikt vidtagande kring minneshantering som gör det både lika säkert som Java och snabbt som C. Scala är ett språk som passar bra till utveckling av DSLer pågrund av dess flexibla syntax och semantik. Den första DSLen är implementerad som en sträng-interpolator. Interpolatorn sammanfogar strängar av Rust kod, under kompileringstid eller exekveringstid, och passerar resultatet till en extern process för statisk kontroll. Den andra DSLen består istället av ett API för att konstruera ett abstrakt syntaxträd, som efteråt kan traverseras och skrivas ut till Rust kod. API:et kombinerar tre koncept: heterogena listor, flytande gränssnitt, och algebraiska datatyper. Dessa tillåter användaren att uttrycka avancerad Rust syntax, såsom polymorfiska strukts, funktioner, och traits, utan att uppoffra typsäkerhet.

Keywords — Kontinuerlig Djup Analys, Domänspeficika Språk, Kodgenerering, Rust, Scala

Contents

1 Introduction **1**

1.1 Background . . . 3
1.2 Problem . . . 3
1.3 Purpose . . . 4
1.4 Goal . . . 4
1.5 Benefits, Ethics and Sustainability . . . 4
1.6 Methodology . . . 5
1.7 Delimitations . . . 5
1.8 Related Work . . . 5
1.8.1 Spark . . . 6
1.8.2 DataFusion . . . 6
1.8.3 Rain . . . 6
1.8.4 ScyllaDB . . . 7
1.8.5 Flare . . . 7
1.8.6 Weld . . . 7
1.8.7 Apache Arrow . . . 8
1.8.8 Voodoo . . . 8
1.9 Outline . . . 9

2 Programming Languages **11**

2.1 Front-end . . . 11
2.1.1 Lexical analysis . . . 11
2.1.2 Syntactic analysis . . . 11
2.1.3 Semantic analysis . . . 12
2.2 Optimizer . . . 13
2.3 Back-end . . . 13
2.4 Domain Specific Languages (DSL) . . . 13

3 The Rust Programming Language **15**

3.1 Basics . . . 15
3.2 Syntax . . . 17
3.3 Ownership . . . 18
3.4 Lifetimes . . . 18
3.5 Types . . . 19
3.6 Unsafe . . . 20
3.7 Compiler overview . . . 21

4 The Scala Programming Language **25**

4.1 Basics . . . 25
4.2 Implicits . . . 26
4.2.1 Type Classes . . . 27
4.2.2 Implicit type inference . . . 27
4.3 Shapeless . . . 28

4.4 Scala DSLs . . . 32
4.4.1 Fluent Interfaces and Method Chaining . . . 32
4.4.2 String literals . . . 32
4.4.3 Language Injection . . . 33
4.4.4 String Interpolation and Quasi Quotation . . . 33
4.4.5 Algebraic Data Types . . . 34
4.4.6 Generalized Algebraic Data Types . . . 35
4.4.7 Tagless Final . . . 35

5 Design **37**
5.1 Shallow Embedded DSL . . . 37
5.2 Deeply Embedded DSL . . . 37
5.2.1 Overview . . . 38
5.3 Emulating `rustc` . . . 38

6 Implementation **40**
6.1 Shallow Embedded DSL . . . 40
6.2 Deeply Embedded DSL . . . 41
6.2.1 Types . . . 42
6.2.2 Literals . . . 44
6.2.3 Unary Operators . . . 44
6.2.4 Binary Operators . . . 45
6.2.5 If-Else . . . 45
6.2.6 Let . . . 46
6.2.7 Functions . . . 46
6.3 Structs . . . 48
6.4 Traits . . . 50

7 Evaluation **51**
7.1 Shallow Embedded DSL - Validation testing . . . 51
7.2 Deeply Embedded DSL - Demo . . . 52

8 Discussion **57**
8.1 Technical details . . . 57
8.2 Design goals . . . 58
8.3 CDA . . . 58

9 Conclusion **61**

References **63**

List of Tables

1 Acronyms used throughout the thesis, listed in alphabetic order. V
2 Static checks of the deeply embedded DSL. 39
3 Rust's operators [74]. 44
4 Results from the validation test. 52

List of Figures

1 An overview of CDA. 2
2 Overview of rustc. 22

List of Listings

1 Struct and tuple struct 16
2 Enum. 16
3 Trait and implementations. 16
4 Closures, struct and enum initialization, method and macro invocation, and pattern matching. 17
5 Rust's statements and expressions. 18
6 Type inference example. 19
7 Type annotations, lifetime annotations, and lifetime elision. 19
8 Class, trait, and val. 25
9 Case class. 26
10 Implicit parameters. 26
11 Implicit conversions 26
12 Implicit classes 27
13 Type class 27
14 Inferring a default type parameter, from. 28
15 Tuple, List and Shapeless HList. 29
16 Polymorphic function 29
17 Comapped example. 30
18 Comapped implementation, from [@HlistOps]. 30
19 Shapeless records. 31
20 Refined 31
21 Fluent interface. 32
22 String literal. 33
23 Language Injection. 33
24 The 's' string interpolator 33
25 Scala quasi-quotes 34
26 ADT example [@GADT]. 34
27 GADT DSL example 35
28 Tagless Final example [@TaglessFinalScala] 36

29 Directory tree of the project. 38
30 Rust string interpolators. 40
31 Runtime string interpolator method. 40
32 Compile time string interpolator macro. 41
33 Interpretations and constructs. 41
34 File implementation. 42
35 Types 42
36 Type class instances for showing types. 43
37 Overriding types. 43
38 Add 44
39 Unary operators. 45
40 Binary operators for assignment and statement termination. 45
41 If-Else definition. 46
42 Let definition. 46
43 Fluent interface parameters (Function) 47
44 Argument definition 47
45 Staging a function argument 47
46 Unstaging a function 48
47 Function example 48
48 Fluent interface parameters - Struct 49
49 Struct field access 49
50 Struct example 49
51 Trait example 50
52 Validation testing of Shallow DSL 51
53 Demo - Types and literals. 52
54 Demo - Struct 52
55 Demo - Trait with method 53
56 Demo - Trait implementation 53
57 Demo - Trait resolution 54
58 Demo - Main function 54
59 Demo - Driver Scala method 54
60 Demo - Generated code 55

Acronyms

Table 1: Acronyms used throughout the thesis, listed in alphabetic order.

Acronym	Definition
ADT	Abstract Data Type
AST	Abstract Syntax Tree
CDA	Continuous Deep Analytics
CSE	Common Sub-expression Elimination
CUDA	Compute Unified Device Architecture
DCE	Dead Code Elimination
DSL	Domain Specific Language
FOAS	First Order Abstract Syntax
GADT	Generalized Abstract Data Type
GPL	General Purpose Language
HIR	High-level IR
HList	Heterogeneous List
HM	Hindley-Milner type inference
HOAS	Higher Order Abstract Syntax
IR	Intermediate Representation
JVM	Java Virtual Machine
LUB	Lower-upper bound
MIR	Mid-level IR
NLL	Non-lexical lifetimes
MPI	Message Passing Interface
OpenCL	Open Computing Language
OpenMP	Open Multi-Processing
RDD	Reliable Distributed Dataset
SAM	Single Abstract Method
SFI	Software Fault Isolation
SQL	Structured Query Language
UAST	Unified AST
UDF	User Defined Function

Acknowledgements

This thesis would not have been completed without support from others. I would first like to thank my examiner, Prof. Christian Schulte, and thesis supervisors, PhD. Lars Kroll, PhD. Paris Carbone and Prof. Seif Haridi, for giving me the opportunity to be part of the CDA project. Over the course of the thesis, their guidance and expert advice have been invaluable. I would also like to thank my peers, Johan Mickos and Oscar Bjuhr, for their insightful comments, brainstorming sessions, and inspiring dedication to their theses. Finally, I would like to thank the Rust community on Discord and Scala community on Gitter for helping me reach a deeper understanding of both languages.

1 Introduction

Deep Analytics, or Big Data Analytics, is the application of data intensive processing techniques in the field of data mining [1]. Data can come from multiple sources in a structured, semi-structured or unstructured format. Continuous Deep Analytics is a new breed of analytics where data is also massive, unbound, and live [2].

This thesis is part of a five year project in collaboration between KTH and RISE SICS to develop a system capable of CDA [2]. The CDA system must be able to run for years without interruption. It also needs to be capable of processing incoming queries in short time windows to support real-time, mission-critical, decision making. CDA is aimed towards both the public sector and industry, much like today's modern general purpose distributed systems. It will enable new time-sensitive applications such as zero-time defense for cyber-attacks, fleet driving and intelligent assistants. These applications involve machine learning and graph analytics, both of which require large scale, data intensive, matrix and tensor computations for affine transformations and convolutional operations [3]. There are two sides to the problem of supporting these kinds of heavy computations: hardware acceleration and computation sharing.

Computation sharing concerns how libraries and languages must work together optimally. Queries to the CDA system may contain user defined functions (UDFs) which appear as black boxes to the pipeline. The aim is to turn the black boxes into white boxes to allow for more fine grained optimizations. Currently, the idea for a solution is to establish common IR across libraries, similar to Weld [4] which provides an IR and runtime. Libraries, including Numpy and TensorFlow, describe their code in Weld's IR and submit it to the Weld runtime. The Weld runtime is then able to merge IRs, and thereby combines the efforts of different libraries.

Hardware acceleration implies the system will exploit the available hardware resources to speedup computation. This is often not an easy task, since developers must have expertise with multiple APIs and programming models which interface with the drivers, e.g., CUDA, OpenCL, OpenMP and MPI [5]. When interfaces change, developers need to update their code. As a further matter, machines in distributed systems can have various hardware configurations. The trend of scaling out, and adding new machines with different hardware, does not make things easier. Hence, hardware acceleration in the presence of hardware heterogeneity becomes an issue of maintenance when code for one machine is neither compatible nor portable to others.

The solution to the problem of hardware heterogeneity is hardware virtualization, which abstracts the physical hardware details away from the user [5]. Spark and Flink realize hardware virtualization through the Java Virtual Machine [6][7]. The JVM is portable, but its support for accelerator architectures, e.g., GPUs, is limited [8]. Hence, the JVM forfeits support for hardware acceleration in favor of support for hardware heterogeneity. Moreover, it also has a runtime overhead, in part owed to garbage collection. To give an example, evaluation by [9] has revealed that a laptop running single threaded low level code can outperform a 128 core Spark cluster in PageRank. High end graph stream processing systems,

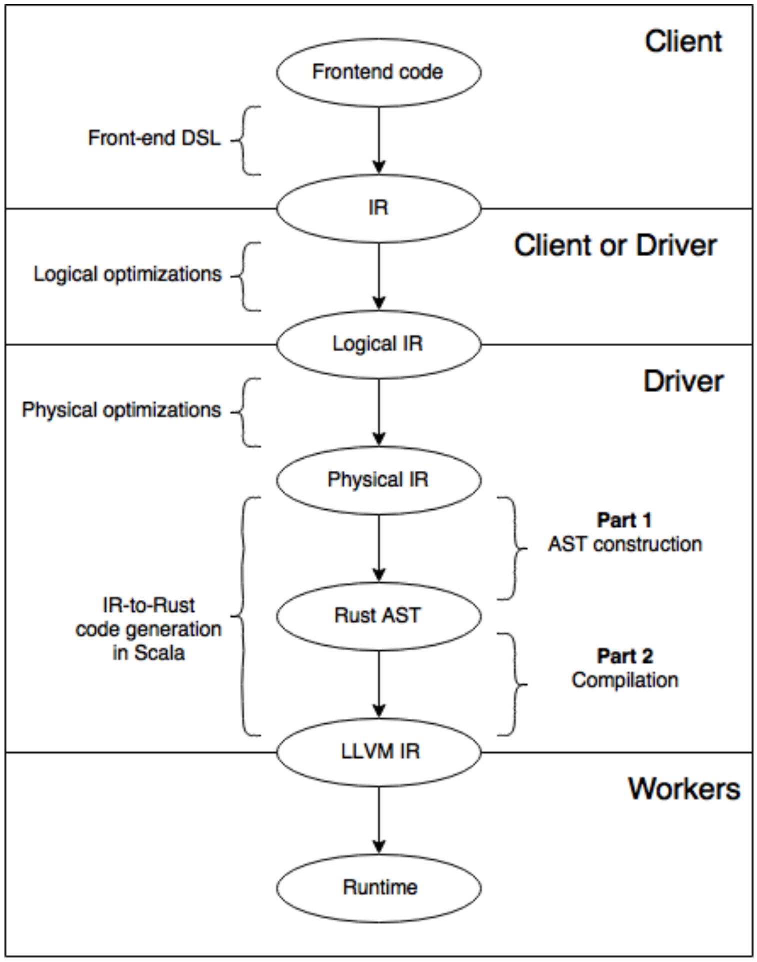

Figure 1: An overview of CDA.

GraphLab and GraphX, were outperformed as well. The evaluation measured 20 PageRank iterations for two medium sized graphs, with the largest being ~105M nodes and ~3.5B edges. An industry standard benchmark by [10] as well identified that Spark SQL spends close to 80% of its execution decoding in-memory data representations. Even when removing this layer of indirection, performance remains 30% slower than hand written C code.

The CDA system will try to obtain both portability and performance simultaneously through code generation. Instead of writing different code for different hardware, the user will write code which is generated for different hardware. Hence, the issue of maintainability is pushed to the developers of CDA rather than the users. An early overview of the system can be viewed in {figure 1}. At the front-end, the user describes the desired behavior of the data processing program in a high level domain-specific language. The front-end code is then transformed into an intermediate representation (IR) containing information about the execution plan and cluster setup. Then, the execution plan is optimized logically through dataflow analysis, and physically by mapping tasks to machines. Next, low level code is generated and compiled for each task, tailored to its machine's hardware. Finally, binaries are deployed in the cluster.

The code generator will also be written as a DSL. Thereby, there are two DSLs involved, the front-end, user-facing, DSL, and the back-end, developer-facing, DSL. This thesis concerns the latter, which will be written as a library in a Scala. The code generator will receive tasks as input and translate low-level source code through the library's interface. How the code is assembled depends on the hardware resources of the machines subject to executing the tasks.

1.1 Background

Domain specific languages are miniature languages, aimed towards a specific problem domain [11][12]. CDA's DSL will be suited to the domain of generating Rust code. DSLs can either be external or embedded. An external DSL is a standalone language, with its own compiler and infrastructure. An embedded DSLs is in contrast implemented as a library in a host language. The embedding can either be deep or shallow. Deep embeddings construct an intermediate representation of the DSL code which can be processed in multiple ways. Shallow embeddings on the other hand directly execute the DSL code as-is.

C and C++ are a commonly used as the language for low-level systems programming [13]. While both compile to fast machine code, neither provide strong safety guarantees. The CDA code generator will therefore instead emit Rust code. Rust is a recent programming language which achieves both safety and performance through a special memory management policy.

Scala will be used as a host language as it is both scalable and naturally suited towards developing embedded DSLs [14]. Programs written in Scala integrate well with Java as both compile down to the byte code and run on the JVM.

1.2 Problem

The problem is thus to implement a DSL for Rust code generation in Scala. It is an important problem to solve, since as of current time, no DSL dedicated to generating Rust code in Scala could be found. Out of interest, two DSLs will be designed and explored, one with a shallow embedding and another with a deep embedding. The following design goals have been set out for both DSLs.

- ***Coverage*** - The DSL should support as much of Rust's syntax and semantics as possible.
- ***Static checking*** - The DSLs should be able to catch errors in the user's code.
- ***Consistency*** - The behavior of the generated program should be consistent with what was specified by the user.
- ***Ease-of-use*** - Writing code in the DSL should be easy, with minimum boilerplate. Developers should also feel some familiarity
- ***Extensibility*** - The DSL should be extensible to the front-end user's for adding new structs and functions, etc.

The problem statement can be defined as: "How do you implement a DSL for Rust code generation in Scala which satisfies the design goals?".

1.3 Purpose

Multiple modern general-purpose distributed systems suffer from performance degradation due to placing workload on the JVM [10]. The purpose of this thesis is to explore code generation as a solution to these problems. Another purpose is to motivate developers to write Rust code generators, instead of C or C++ code generators. As a result, future distributed systems might become even more secure and reliable.

1.4 Goal

The goal of the thesis is to develop a prototype back-end code-generator for the CDA project by exploring DSLs, Rust and Scala. Developers of future distributed systems may benefit from the discoveries. The following deliverables are expected:

- A background study of programming languages, Rust, Scala, and the theory behind DSLs.
- Two Scala DSLs for Rust code generation, and a description of their design and implementation.
- An evaluation of the DSLs, taking the design goals into consideration.

1.5 Benefits, Ethics and Sustainability

CDA will improve upon existing state-of-the-art systems like Spark and Flink. Flink is being used by large companies including Alibaba, Ericsson, Huawei, King, LINE, Netflix, Uber, and Zalando [15]. Since performance is a crucial metric, these companies may benefit from incorporating CDA into their business. As an example, Alibaba uses Flink for optimizing search rankings in real-time. CDA may allow for more complex and data intensive search rank optimizations. This should benefit the customer, whom will have an easier time finding and buying the right product, profiting the company. Whether the economic growth is sustainable or not depends on the context. For example, if more non-recyclable products are being sold, it might negatively impact the environmental sustainability through increased pollution, which is not economically sustainable. The impact might not be as severe if instead digital products are being sold.

CDA's power however comes with a responsibility, as it can be used to either help, or harm others. Although one of the use cases for CDA is cyber defence, there is nothing preventing it from being used for the opposite. Another concern is how CDA's possible contribution to artificial intelligence might impact social sustainability. With more computation power comes better trained AI. This could lead to more positions of employment being succumbed by artificial intelligence, e.g., intelligent assistants and driverless vehicles.

In this thesis' perspective, it crucial that the DSL is statically safe and does not generate buggy code which could compromise security. The behavior of the generated code should be what was specified in the IR. Furthermore, low level code is able to utilize hardware with greater efficiency than high level code. Thus, better performance in might implicate less waste of resources, which could be sustainable for the environment.

1.6 Methodology

The first step of the thesis was to do a background study of work related to distributed systems, code generation, and DSLs. The purpose was to gain insight into the problem which the thesis addresses and also explore existing solutions. The study provided an overview of how code generation DSLs can enhance performance of distributed systems. It also gave an idea of the design goals which might be desirable for for the CDA code generation DSL. Afterwards, the objective was to study Rust and Scala, through reading the documentations, practicing, and talking with the community. The intent was to gain a deep enough understanding to be able to design and implement a DSL in Scala for generating Rust code. Learning Rust went fast, since it only required studying the syntax and semantics. Scala took more time as it also involved learning the Shapeless library and an assortment of prominent programming patterns. A majority of the DSLs, from papers and other resources, were written in Haskell. Haskell could be considered the lingua franca of functional programming, and was thereby also necessary to learn at a basic level. Next was the design and implementation. The supervisors, Lars Kroll and Paris Carbone, had initially suggested different design approaches for the CDA code generation DSL. Using these approaches as a starting point, two DSLs were prototyped. Both DSLs underwent multiple revisions before settling on the final version. The evaluation involved testing the DSLs with respect to the design goals. Following the structure of other DSL papers, demoing and validation testing were chosen as the evaluation methods. Conclusively, most of the thesis was spent on reading papers and writing the report. The remaining portion was spent on implementing the DSLs.

1.7 Delimitations

Only Rust will be used as the target language for code generation. It would be interesting to embed other languages, e.g., C and C++, but this is out of scope for the thesis. The DSL also aims support many, but not all of Rust's features. Instead, the focus is quality over quantity. Programs written in the DSL are primarily meant to glue pieces of code together from different Rust libraries. Hence, the programs will be relatively small, and take short time to compile and run. For this reason, performance testing was excluded from the evaluation.

1.8 Related Work

This section gives a brief overview of work related to CDA, beginning with an introduction to Spark, DataFusion, Rain, and ScyllaDB, which are modern distributed systems. Then, Flare, Weld, Apache Arrow, and Voodoo which are different approaches to improving the performance of distributed systems, are described.

1.8.1 Spark

Spark is a modern general purpose distributed system for batch processing [6]. It was designed to get around the limitations of MapReduce. While MapReduce is able to perform large-scale computations on commodity clusters, it has an acyclic dataflow model which limits its number of applications. Iterative applications such as most machine learning algorithms, and interactive analytics are not feasible on MapReduce. Spark is able to support these features, while retaining the scalability and reliability of MapReduce. The core abstraction of Spark is a Resilient Distributed Dataset (RDD). A RDD is a read-only collection of objects partitioned over a cluster. The RDD stores its lineage, i.e., the operations which were applied to it, which lets it re-build lost partitions.

RDDs support two forms of operations: transformations and actions [16]. Transformations, e.g., map, and foreach, transform an RDD into a new RDD. Actions, e.g., reduce, and collect, returns the RDD's data to the driver program. All transformations are lazily evaluated. With lazy evaluation, data in a computation is materialized only when necessary. This speeds up performance by reducing the data movement overhead [4].

Spark SQL is an extension to Spark which brings support for relational queries [17]. It introduces a DataFrame abstraction. Whereas RDDs are a collection of objects, DataFrames are a collection of records. DataFrames can be manipulated both with Spark's standard procedural API and with a new relational API. The relational API supports SQL written queries.

1.8.2 DataFusion

DataFusion is a distributed computational platform which acts as a proof-of-concept for what Spark could be if it were to be re-implemented in Rust [18]. Spark's scalability and performance is challenged by the overhead of garbage collection and Java object serialization. While Tungsten addresses these issues by storing data off-heap, they could be avoided altogether by transitioning away from the JVM. DataFusion provides functionality which is similar to Spark's SQL's DataFrame API, and takes advantage of the Apache Arrow memory format. DataFusion outperforms Spark for small datasets, and is still several times faster than Spark when computation gets I/O bound. In addition, DataFusion uses less memory, and does not suffer from unforeseen garbage collection pauses or OutOfMemory exceptions.

1.8.3 Rain

Rain is an open source distributed computational framework, with a core written in Rust and an API written in Python [19]. Rain aims to lower the entry barrier to the field of distributed computing by being portable, scalable and easy to use.Computation is defined as a task-based pipeline through dataflow programming. Tasks are coordinated by a server, and executed by workers which communicate over direct connections. Workers may also spawn subworkers as local processes. Tasks are either BIFs or UDFs. UDFs can execute Python code, and can make calls to external applications. Support for running tasks as plain C code, without having to link against Rain, is on the roadmap.

1.8.4 ScyllaDB

NoSQL is a new generation of high performance data management systems for Big Data applications [20]. The consistency properties of relational SQL systems limit their scalability options. In contrast, NoSQL systems are more scalable since they store data in flexible and replicable formats such as key-value pairs. One of the leading NoSQL data stores is Cassandra, which was originally developed by Facebook. Cassandra is written in Java and provides a customizable and decentralized architecture. ScyllaDB is an open-source re-write of Cassandra into C++ code with focus on utilization of multi-core architectures, and removing the JVM overhead.

Most of Cassandra's logic stays the same in ScyllaDB. Although, one notable difference is their caching mechanisms. Caching reduces the disk seeks of read operations. This helps decrease the I/O load which can be a major bottleneck in distributed storage systems. Cassandra's cache is static while ScyllaDB's cache is dynamic. ScyllaDB will allocate all available memory to its cache and dynamically evict entries whenever other tasks require more memory. Cassandra does not have this control since memory is managed by the JVM garbage collector. In evaluation, ScyllaDB's caching strategy improved the reading performance by less cache misses, but also had a negative impact on write performance.

1.8.5 Flare

CDA's approach to code generation draws inspiration from Flare which is an alternate back-end to Spark [10]. Flare bypasses Spark's abstraction layers by compiling queries to native code, replacing parts of the Spark runtime, and by extending the scope of optimizations and code generation to UDFs. Flare is built on top of Delite which is a compiler framework for high performance DSLs, and LMS, a generative programming technique. When applying Flare, Spark's query performance improves and becomes equivalent to HyPer, which is one of the fastest SQL engines.

1.8.6 Weld

Libraries are naturally modular: they take input from main memory, process it, and write it back [4]. As a side effect, successive calls to functions of different libraries might require materialization of intermediate results, and hinder lazy evaluation.

Weld solves these problems by providing a common interface between libraries. Libraries submit their computations in IR code to a lazily-evaluated runtime API. The runtime dynamically compiles the IR code fragments and applies various optimizations such as loop tiling, loop fusion, vectorization and common sub-expression elimination. The IR is minimalistic with only two abstractions: builders and loops. Builders are able to construct and materialize data, without knowledge of the underlying hardware. Loops consume a set of builders, apply an operation, and produce a new set of builders. By optimizing the data movement, Weld is able to speedup programs using Spark SQL, NumPy, Pandas and Tensorflow by at least 2.5x.

1.8.7 Apache Arrow

Systems and libraries like Spark, Cassandra, and Pandas have their own internal memory format [21]. When transferring data from one system to another, about 70-80% of the time is wasted on serialization and deserialization. Apache Arrow eliminates this overhead through a common in-memory data layer. Data is stored in a columnar format, for locality, which maps well to SIMD operations. Arrow is available as a cross-language framework for Java, C, C++, Python, JavaScript, and Ruby. It is currently supported by 13 large open source projects, including Spark, Cassandra, Pandas, Hadoop, and Spark.

1.8.8 Voodoo

Voodoo is a code generation framework which serves as the backend for MonetDB [22]. MonetDB is a high performance query processing engine. Voodoo provides a declarative intermediate algebra which abstracts away details of the underlying hardware. It is able to express advanced programming techniques such as cache conscious processing in few lines of code. The output is optimized OpenCL code.

Code generation is complex. Different hardware architectures have different ways of achieving performance. Moreover, the performance of a program depends on the input data, e.g., for making accurate branch predictions. As a result, code generators need to encode knowledge both about hardware and data to achieve good performance. In reality, most code generators are designed to generate code solely for a specific target hardware. Voodoo solves this through providing an IR which is portable to different hardware targets. It is expressive in that it can be tuned to capture hardware-specific optimizations of the target architecture, e.g., data structure layouts and parallelism strategies. Additional defining characteristics of the Voodoo language are that it is vector oriented, declarative, minimal, deterministic and explicit. Vector oriented implicates that data is stored in the form of vectors, which conform to common parallelism patterns. By being declarative, Voodoo describes the dataflow, rather than its complex underlying logic. It is minimal in that it consists of non-redundant stateless operators. It is deterministic, i.e., it has no control-flow statements, since this is expensive when running SIMD unit parallelism. By being explicit, the behavior of a Voodoo program for a given architecture becomes transparent to the front end developer.

Voodoo is able to obtain high parallel performance on multiple platforms through a concept named controlled folding. Controlled folding folds a sequence of values into a set of partitions using an operator. The mapping between value and partition is stored in a control vector. High performance is achieved by executing sequences of operators in parallel. Voodoo provides a rich set of operators for controlled folding which are implemented in OpenCL. Different implementations for the operators can be provided depending on the backend.

When generating code, Voodoo assigns an Extent and Intent value to each code fragment. Extent is the code's degree of parallelism while Intent is the number of sequential iterations per parallel work-unit. These factors are derived from the control vectors and optimize the performance of the generated program.

1.9 Outline

The coming chapter gives a bird's eye view over the area of programming languages and domain specific languages. Embedding one language in another requires in-depth knowledge about how both languages operate. Chapter 3 therefore covers Rust's syntax and semantics, and aims to answer the question of what is needed to embed Rust in another language. Chapter 4 sheds light on Scala's features and modern approaches for embedding DSLs, to identify if Scala can meet Rust's demands. The design of the Rust DSLs is covered in chapter 5. Chapter 6 contains implementation details, and chapter 7 evaluates the implementations, with validation testing and a demonstration. Chapter 8 discusses the results with respect to technical details, design goals, and CDA. Finally, section 9 reflects over how the project went, and considers what there is to in future work.

2 Programming Languages

A programming language is a tool for communicating with computers [23]. Although programming languages can have vastly different designs, their implementations, to some degree, follow the same structure. Each program starts off as source code and is analyzed and transformed progressively in a series of stages. Compilers typically divide the stages into three components, a front-end, optimizer, and back-end [24].

2.1 Front-end

The front-end statically verifies the lexical, syntactic and semantic correctness of the program. These are analogous to verifying that sentences in natural language contain correctly spelled words, are grammatically correct, and have a sensible meaning. Simultaneously, the front-end will transform the program into an IR which is at a higher level abstraction and is easier to work with.

2.1.1 Lexical analysis

First, source code is scanned into a flat stream of tokens according to a regular expression. The source code is composed of whitespace and lexemes, i.e., lexical units, which are the words of the programming language [25]. Some languages have a pre-processor which operates before the lexer. The pre-processor may alter the source code through macro expansion and various directives [26]. Tokens are categories of lexemes, including identifiers, keywords, literals, separators and operators. These resemble nouns, verbs, and adjectives. The regular expression could be viewed as a lexicon or vocabulary which defines tokens and lexemes for a regular language. Some tokens, such as identifiers and literals, also have an associated semantic value.

2.1.2 Syntactic analysis

After scanning, tokens are parsed into a parse tree according to a grammar. The parse tree describes the concrete syntactic structure of the program. Nodes in the tree are syntactic units, referred to as symbols. Each symbol is classified as either terminal or non-terminal. While terminals are tokens, non-terminals are symbols which may be substituted by zero or more symbols and are thereby analogous to phrases. The grammar is in the form of production rules which dictate which symbols substitute other symbols. Grammars for programming languages are in general context-free. In other words, how a symbol is parsed does not depend on its relative position to other symbols. In contrast, only a subset of the natural languages are assumed to have context-free grammars [27]. History has shown however that the dissimilarity between programming languages and natural languages is decreasing as programming languages are becoming increasingly more high level [28].

A parser can be implemented by hand with a parser combinator or be generated with a parser generator [28, Ch. 3]. Parser combinators are DSLs for constructing backtracking recursive descent parsers which operate at runtime. In contrast, parser generators generate a parser according to a context-free grammar, e.g., Backus-Naur form (BNF) grammar. Generated parsers can sometimes be more restrictive than parser combinators, but are more predictable, and offer greater performance. In addition, some parser generators can generate to multiple target languages.

After parsing, the parse tree is converted into an abstract syntax tree (AST) [28, p. 33]. The AST excludes needless information such as the appearance and ordering of symbols. Although each language parses to a different AST, the concept of a unified AST (UAST) across different languages and paradigms has been explored in the field of code refactoring [29]. Languages would parse into separate ASTs and then combine into a UAST which enables cross-language optimizations. The idea has been put into practice in the ParaPhrase project which coalesces Erlang, C and C++ under a UAST. ParaPhrase then applies cross-language refactorings to the UAST, introducing parallel programming patterns to sequential code.

2.1.3 Semantic analysis

The remaining stages of the front-end concern the program's semantics [23]. The compiler has to determine the meaning of the program and verify if it is sensible. What constitutes as semantics varies between languages. Most languages start with *name-binding* which binds each symbol with an identifier to the site where the identifier was first introduced.

Another central part of semantic analysis is often *type checking*, which concerns catching inconsistencies in the program. Type checking involves verifying that each operator has matching operands, to for example prevent an integer from being multiplied with a string [30]. In some languages, it is however possible to add values of different types by coercing, i.e., implicitly converting, one type to the other. The set of rules which types must conform to are defined by the *type system*. These rules are sometimes notated as *inference rules*, which draw *conclusions* from *premises*. A conclusion or premise is a *judgment* in the form of $e\colon T$, meaning e has type T [28]. For example, given the premises $x\colon float$ and $y\colon float$, it can be concluded that $x{+}y\colon float$. Thus, given an expression e and a type T, type checking must decide whether $e\colon T$. Some type systems also support type inference, which is about finding a type T for e such that $e\colon T$. How a type is inferred depends on how it is used. For example, if $x\colon float$ and $x{+}y\colon int$, then $y\colon int$ could be a possible solution.

The isomorphism between type systems and logic systems is referred to as the Curry-Howard isomorphism [31]. Type systems derive two properties from logic systems, *soundness* and *completeness* [32]. A sound type system will reject programs which are not type safe. A complete type system will not reject programs which are type safe. The former prevents false negatives (misses) and the latter false positives (false alarms). Type systems of modern languages are sound but not complete [33]. If a type system is unsound for some property and moves the responsibility for checking it to the programmer, then it is weakly typed. Conversely, if a type system is statically sound, or is unsound for some property but employs dynamic checks to prevent it, then it is strongly typed. Some compilers store type information about AST symbols in a *symbol table* [23]. The parser can also leave empty fields in the AST which get filled in later, known as *attributes*. Languages without type checking trade type errors for runtime errors and are called *untyped* languages.

2.2 Optimizer

Optimizations commence when the program is at a sufficiently high level of abstraction [23]. The optimizer applies various target-independent optimizations to the IR such as loop unrolling, dead-code elimination (DCE), and common sub-expression elimination (CSE). Loop unrolling unrolls a loop into a larger loop body with fewer iterations, allowing for better instruction scheduling [30, Ch. 9]. DCE removes code that computes values which never gets used. CSE locates expressions that evaluate to the same value, and substitutes them with a common variable only needing to be computed once. Certain optimizations have a space-time tradeoff. For example, loop unrolling produces faster code, but also increases the size of the executable.

2.3 Back-end

Finally, the back-end synthesizes machine code for a specific architecture [23]. Multiple programming languages re-use the same back-end through a shared IR. Another option is to *transpile* the program. Instead of generating IR code targeting some back-end, transpilers generate code for some other programming language. Afterwards, a compiler for the target language may compile the generated code. Another approach is to *interpret*, i.e., directly execute, the program. Some interpreters execute by recursively traversing the AST, these are referred to as tree-walk interpreters. Certain compilers produce hardware-independent byte code instead, of hardware-dependent machine code, which gets interpreted by a *virtual machine* (VM). Some VMs support just-in-time (JIT) compilation where byte-code is compiled down to machine code just before being executed. This is combined with profiling, i.e., inspecting the code as it is being run, to allow for runtime optimizations. Ahead-of-time compilation only synthesizes code once, i.e., without runtime optimizations.

2.4 Domain Specific Languages (DSL)

There are two categories of programming languages: General Purpose Languages (GPLs) and Domain Specific Languages (DSLs). DSLs are small languages suited to interfacing with a specific problem domain [12] and often act as a complement to GPLs. In contrast, GPLs are designed for a wide variety of problem domains. GPLs are, unlike DSLs, always Turing complete. Therefore, anything that can be programmed in a DSL can also be programmed in a GPL. The opposite may not always apply. Using DSLs can nevertheless lighten the burden of solving specific problems. For example, SQL is convenient for writing search queries on relational data. By being restricted to a certain problem domain, DSLs can offer high level abstractions without sacrificing performance. DSLs are also capable of aggressive domain-specific optimizations.

A DSL can either be *external* or *embedded*. External DSLs exist in their own ecosystem, with a custom compiler, debugger, editor, etc. Building and maintaining these tools can be cumbersome. In contrast, embedded DSLs reside within a host GPL as a library or framework. As a result, they take less time to develop, but are restricted in their expressiveness by the host GPL's syntax and semantics.

Embedded DSLs can either have a *shallow* or *deep* embedding [12][11]. A shallow embedding implicates the DSL is executed eagerly without constructing an IR. A deep embedding means the DSL creates an IR which can be interpreted in multiple ways, e.g., generated, optimized, compiled, and checked. The host GPL acts as a metalanguage and the DSL as an object language. The metalanguage is able to re-shape the object language, since it is only data.

A powerful IR is higher order abstract syntax (HOAS), which is a generalization of the ordinary first-order abstract syntax (FOAS) that compilers use to encode ASTs [34]. In FOAS, nodes refer to each other through the use of symbolic identifiers. HOAS generalizes FOAS by capturing name binding information. Hence, nodes in HOAS refer directly to each other with links, forming an *abstract syntax graph* instead of a tree.

Embedded DSLs come in many flavors. Popular approaches to embedding DSLs, specifically in Scala, are fluent interfaces, Quasi-Quotation, Generalized Abstract Data Types (GADTs), and Tagless Final. How these function is covered in section 4.

3 The Rust Programming Language

C and C++ have for decades been the preferred languages for low level systems programming [35]. Both offer excellent performance, but are also unsafe. Mistakes in pointer aliasing, pointer arithmetic and type casting, leading to memory violations, can be hard to detect, even for advanced software verification tools. Recurrent errors are memory leaks, null pointer dereferences, segmentation faults, and data races. Although C++ facilitates countermeasures, e.g., smart pointers, RAII, and move semantics, its type system is too weak to statically enforce their usage [13]. Meanwhile, safe high-level languages like Java solve the safety issues through managed runtime coupled with a garbage collector. This safety comes at a cost since garbage collection incurs a big overhead. Overcoming the tradeoff between safety and control has long been viewed as a holy grail in programming languages research.

Rust is a modern programming language conceived and sponsored by Mozilla [13]. It overcomes the tradeoff between safety and control through a compile time memory management policy based on ownership, unique references, and lifetimes. Ownership prevents double free errors, unique references prevent data races, and lifetimes prevent dangling pointers. In addition, Rust offers zero-cost abstractions such as pattern matching, generics, traits, higher order functions, and type inference.

Packages, e.g., binaries and libraries, in Rust are referred to as crates [36, Ch. 4]. Cargo is a crate manager for Rust which can download, build, and publish crates. A large collection of open-source crates can be browsed at.[1] One of the largest crates to date is the Servo browser engine, developed by Mozilla. Servo's strict demands for security and memory safe concurrency have attributed to shaping Rust into what it is today [37, Ch. 1].

Rust has a stable, beta, and nightly build [38, Ch. 4]. The nightly build is updated on a daily-basis with new experimental features. Once every six weeks, the latest nightly build is promoted to beta. After six additional weeks of testing, beta becomes stable. Since Rust's original release, there have been multiple major revisions. Dropped features include a typestate system [39], and a runtime with green threaded-abstractions [40].

3.1 Basics

A Rust crate is a hierarchy of modules [38, Ch. 3]. Modules contain structs, traits, methods, enums, etc., collectively referred to as items. Items support parametric polymorphism, i.e., generics. Listing 48 defines a `Rectangle` struct and a `Triangle` tuple struct with fields of generic type `T`. Structs encase related values, and tuple structs are a variation of structs with unnamed fields.

[1] `https://www.crates.io`.

Listing 1 Struct and tuple struct

```
struct Rectangle<T> {
  width:  T,
  height: T,
}
struct Triangle<T>(T, T, T);
```

Enums are tagged unions which can wrap values of different types. For example, `Shape` in listing 2 wraps values of type `Rectangle` and `Circle`.

Listing 2 Enum.

```
enum Shape<T> {
  Rectangle(Rectangle<T>),
  Triangle(Triangle<T>),
}
```

Traits define methods for an abstract type `Self`, and are implemented in ad-hoc fashion, comparable to type classes in other programming languages. In listing 51, `Geometry` is a trait which defines a method for calculating the `perimeter`. `Rectangle`, `Triangle` and `Shape` implement the `Geometry` trait. Functions return the last expression in the function body, and as a result not require an explicit `return` statement.

Listing 3 Trait and implementations.

```
trait Geometry<T> {
  fn perimeter(&self) -> T;
}
impl<T: Add<Output=T>+Copy> Geometry<T> for Rectangle<T> {
  fn perimeter(&self) -> T {
    self.width + self.width + self.height + self.height
  }
}
impl<T: Add<Output=T>+Copy> Geometry<T> for Triangle<T> {
  fn perimeter(&self) -> T {
    self.0 + self.1 + self.2
  }
}
impl<T: Add<Output=T>+Copy> Geometry<T> for Shape<T> {
  fn perimeter(&self) -> T {
    match self {
      &Shape::Rectangle(ref r) => r.perimeter(),
      &Shape::Triangle(ref t)  => t.perimeter(),
    }
  }
}
```

Note how the implementations require traits to be implemented for the generic types. `T: Add<Output=T>` requires a trait for addition to be implemented for `T`. `Output=T` implicates the result of the addition is of type `T` and `Copy` permits `T` to be copied. In the implementation for `Shape`, a **`match`** expression is used to unwrap the enum into references of its values.

Listing 4 defines the main function for testing the code. First, a closure, i.e., lambda function, `calc` is defined for calculating and printing the perimeter of a shape. It takes a `kind` argument, indicating whether the shape is a `Rectangle` or `Triangle`, and an array `v` storing the sides of the shape. The '`!`' in `println!` indicates `println` is a macro and not a method.

Listing 4 Closures, struct and enum initialization, method and macro invocation, and pattern matching.

```
fn main() {
  let calc = |kind: &str, v: &[i32]| {
    let shape = match kind {
      "Rectangle" => Shape::Rectangle(Rectangle{ width: v[0], height: v[1] }),
      "Triangle"  => Shape::Triangle(Triangle( v[0], v[1], v[2])),
      _           => std::process::exit(-1)
    };
    println!("Perimeter of {} is {}", kind, shape.perimeter());
  };
  calc("Rectangle", &[5,7]);  // Perimeter of Rectangle is 24
  calc("Triangle", &[3,3,3]); // Perimeter of Triangle is 9
}
```

3.2 Syntax

Rust's syntax is mainly composed of *expressions*, and secondarily *statements* [36, Ch. 6]. Expressions evaluate to a value, may contain operands, i.e., sub-expressions, and can either be mutable or immutable. Unlike C, Rust's control flow constructs are expressions, and can thereby be *side-effect* free. For instance, loops can return a value through the break statement. Expressions are either *place expressions* or *value expressions*, commonly referred to as *lvalues* and *rvalues* respectively. Place expressions represent a memory location, e.g., array an indexing, field access, or dereferencing operation, and can be assigned to if mutable. Value expressions represent pure values, e.g., literals, and can only be evaluated.

Statements are divided into *declaration statements* and *expression statements*. A declaration statement introduces a new name for a variable or item into a namespace. Variables are by default declared immutable, and are visible until end of scope. Items are components, e.g., enums, structs and functions, belonging to a crate. Expression statements are expressions which evaluate to the unit type by ignoring their operands' return results, and in consequence only produce side-effects. Listing 5 displays examples of various statements and expressions.

Listing 5 Rust's statements and expressions.

```
struct Foo;                 // Item declaration
let foo = Foo;              // Let declaration
loop { v.pop(); break; }    // Expression statement
loop { break v.pop(); }     // Value expression
(1+2)                       // Value expression
*(&mut (1+2))               // Place expression
foo                         // Place expression
```

3.3 Ownership

When a variable in Rust is bound to a resource, it takes *ownership* of that resource [35]. The owner has exclusive access to the resource and is responsible for dropping, i.e., de-allocating, it. Ownership can be *moved* to a new variable, which in consequence breaks the original binding. Alternatively, the resource can be *copied* to a new variable, which results in a new ownership binding. Variables may also temporarily *borrow* a resource by taking a reference of it. The resource can either be mutably borrowed by at most one variable, or immutably borrowed by any number of variables. Thus, a resource cannot be both mutably and immutably borrowed simultaneously. The concept of ownership and move semantics relates to affine type systems wherein every variable can be used at most once [41].

Ownership prevents common errors found in other low level languages such as double-free errors, i.e., freeing the same memory twice. Moreover, the borrowing rules eliminate the risk of data-races. Although Rust is not the first language to adopt ownership, previous attempts were generally restrictive and demanded verbose annotations [13]. Rust's ownership is able to solve complex security concerns such as Software Fault Isolation (SFI) and Static Information Control (IFC) [35].

SFI enforces safe boundaries between software modules that may share the same memory space, without depending on hardware protection. If data is sent from a module, then only the receiver should be able to access it. This can get complicated when sending references rather than values in languages without restrictions to mutable aliasing. Rust's ownership policy ensures that the sent reference cannot be modified by the sender while it is borrowed by the receiver.

IFC imposes confidentiality by tracing information routes of confidential data. This becomes very complex in languages like C where aliasing can explode the number of information routes. IFC is easier in Rust because it is always clear which variables have read or write access to the data.

3.4 Lifetimes

Every resource and reference has a lifetime which corresponds to the time when it can be used [42][43]. The lifetime of a resource ends when its owner goes out of scope, and in consequence causes the resource to be dropped. Lifetimes for references can in contrast exceed their borrower's scope, but not their its referent's. A reference's lifetime can also be tied to others' [41]. For instance, a reference A to a reference B imposes the constraint that the lifetime of A must live for at least as long as the lifetime of B. Without this constraint, A might eventually become a dangling pointer, referencing freed memory.

Rust has a powerful type and lifetime inference which is local to function bodies. Listing 6 displays how Rust is able to infer the type of a variable based on information past its declaration site.

Listing 6 Type inference example.

```rust
fn foo() {
  let x = 3;
  let y: i32 = x + 5;
  let z: i64 = x + 5; // Mismatched types
}                     // Expected i64, found i32
```

Since the inference is not global, types and lifetimes must be annotated in item signatures as illustrated in listing 7[38, Ch. 3]. Lifetimes in function signatures can however occasionally be concluded with a separate algorithm named *lifetime elision*. Lifetime elision adopts three rules. First, every elided lifetime gets a distinct lifetime. If a function has exactly one input lifetime, that lifetime gets assigned to all elided output lifetimes. If a function has a self-reference lifetime, that lifetime gets assigned to all elided output lifetimes. Cases when the function signature is ambiguous and the rules are insufficient to elide the lifetimes demand explicit lifetime annotations.

Listing 7 Type annotations, lifetime annotations, and lifetime elision.

```rust
fn bar(x, y) -> _ { x }            // Does not compile
fn bar<T>(x: T, y: T) -> T { x } // Compiles

fn baz<T>(x: &T, y: &T) -> &T { x }              // Does not compile
fn baz<'a,T>(x: &'a T, y: &'a T) -> &'a T { x } // Compiles

fn qux<T>(x: &T) -> &T { x } // Compiles (Lifetime elision)
```

3.5 Types

Rust has primitive, nominal, structural, pointer, function pointers, and closure types [36, Ch. 7]. Primitive types include integers, floats, booleans, textual types, and the never type. Structs, unions and enums are nominal types. Nominal types can be recursive and generic. Arrays, tuples and slices are structural types, and cannot be recursive. Pointers are either shared references, mutable references, raw pointers or smart pointers. Function pointers identify a function by its input and output types. Closures have types as well, but hidden from the user.

There exists support for subtyping of lifetimes, but not structs [44]. Naturally, it should be possible to use a subtype in place of its supertype. In the same sense, it should be possible to use a long lifetime in place of a shorter one. Hence, a lifetime is a subtype of another if the former lives for at least as long as the latter. Type theory formally denotes subtyping relationships by `<:`, e.g., `A <: B` indicates `A` is a subtype of `B`.

Rust's type system includes *type constructors* [44]. A type constructor is a type which takes type parameters as input and returns a type as output, e.g., a generic nominal type `Option<T>` or pointer type `&'a mut T`. Types which take no type parameters are *proper types*. Type constructors can be covariant, contravariant, or invariant over their input. If `T <: U` implies `F<T> <: F<U>`, then `F` is covariant over its input. If `T <: U` implies `F<U> <: F<T>`, then `F` is contravariant over its input. `F` is invariant over its input if no subtype relation is implied. Immutable references are covariant over both lifetime and type, e.g., `&'a T` can be coerced into `&'b U` if `'a <: 'b` and `T <: U`. Contrarily, mutable references are variant over lifetime, but invariant over type. If type was covariant, then a mutable reference `&'a mut T` could be overwritten by another `&'b mut U`, where `'a <: 'b` and `T <: U`. In this case, `&'a` would eventually become a dangling pointer.

3.6 Unsafe

Ownership and borrowing rules can in some cases be restrictive, specifically when trying to implement cyclic data structures [45][35]. For instance, implementing doubly-linked lists, where each node has a mutable alias of its successor and predecessor is difficult. There are in general two ways to achieve mutable aliasing. The first way is to use a reference counter (`Rc<T>`) together with interior mutability (`RefCell<T>`). The reference counter, i.e., smart pointer, allows a value to be immutably owned by multiple variables simultaneously. A value's reference counter is incremented whenever a new ownership binding is made, and decremented when one is released. If the counter reaches zero, the value is de-allocated. Interior mutability lets a value be mutated even when there exists immutable references to it. It works by wrapping a value inside a `RefCell`. Variables with a mutable or immutable reference to the `RefCell` can then mutably borrow the wrapped value. By combining reference counting with interior mutability, i.e., `Rc<RefCell<T>>`, multiple variables can own the `RefCell` immutably, and are able to mutably borrow the value inside.

The other way of achieving mutable aliasing is through unsafe blocks [45][35]. Unsafe blocks are blocks of code wherein raw pointers can be dereferenced. Raw pointers are equivalent to C-pointers, i.e., pointers without any safety guarantees. Multiple raw pointers can point to the same memory address. The compiler cannot verify the static safety of unsafe blocks. Therefore, code inside these blocks have the potential to cause segmentation faults or other undefined behavior, and should be written with caution. While Rust is safe without using unsafe operations, many Rust libraries including the standard library, use unsafe operations. Unsafe blocks are primarily used for making external calls to C. The support for calling C++ from Rust is limited however. RustBelt is an extension to Rust which verifies the soundness of unsafe blocks [13]. It builds a semantic model of the language which is then verified against typing rules. A Rust program with well-typed unsafe blocks should not express any undefined behavior.

3.7 Compiler overview

Rust's primary compiler is *rustc* [46]. An overview of the pipeline for compiling source code into machine code is illustrated in figure 2. The overview is based on the rustc compiler guide, which is currently under development [46]. Hence, all parts of rustc are not covered in the overview, and certain concepts might be overly abstract. As Rust is also rapidly evolving, some parts may eventually be outdated.

Lexing Rust's lexer distinguishes itself from other lexers in how its output stream of tokens is not flat, but nested [46, Ch. 10]. Separators, i.e., paired parentheses '`()`', braces '`{}`', and brackets '`[]`', form token trees. Token trees are an essential part of the macro system. As a by-product, mismatched separators are among the first errors to be caught by the front-end. The lexer will also scan for raw string literals [47]. In normal string literals, special characters need to be escaped by a backslash, e.g., `" \" "`. Rust string literals can instead be annotated as raw, e.g., `r#" " "#`, which allows ommitting the backslash. For a string literal to be raw, it must be surrounded by more hashes than what it contains, e.g., `r#"##"#` would need to be rewritten to `r##"##"##`. The implication is that Rust's lexical grammar is neither regular nor context free as scanning raw string literals requires context about the number of hashes. For this reason, the lexer is hand written as opposed to generated.

Parsing Rust's parser is a recursive descent parser, handwritten for flexibility [46, Ch. 10]. A non-canonical grammar for Rust is available in the repository [48]. The lexer and parser can be generated with *flex* and *bison* respectively. While bison generates parsers for C, C++ and Java, flex only targets C and C++ [49][50]. JFlex is however a close alternative to Flex which targets Java [51]. The parser produces an AST as output that is subject to macro expansion, name resolution, and configuration. Rust's AST is atypical as it preserves information about the ordering and appearance of nodes. This sort of information is commonly stored in the parse tree and stripped when transforming into the AST.

Macro expansion Rust's macros are at a higher level of abstraction compared to standard C-style macros which operate on raw bytes in the source files [46, Ch. 11]. Macros in Rust may contain meta-variables. Whereas ordinary variables bind to values, meta-variables bind to token-trees. Macro expansion expands macro invocations into the AST, according to their definitions, and binds their meta-variables to token-trees. This task is commissioned to a separate regex-based macro parser. The AST parser delegates any macro definition and invocation it encounters to the macro parser. Conversely, the macro-parser will consult the AST parser when it needs to bind a meta-variable.

Configuration Item declarations can be prepended by an attribute which specifies how the item should be treated by the compiler. A category of attributes named *conditional compilation attributes* are resolved alongside macro expansion [46, Ch. 7][38, Ch. 4]. Other are reserved for later stages of compilation. A conditional compilation attribute

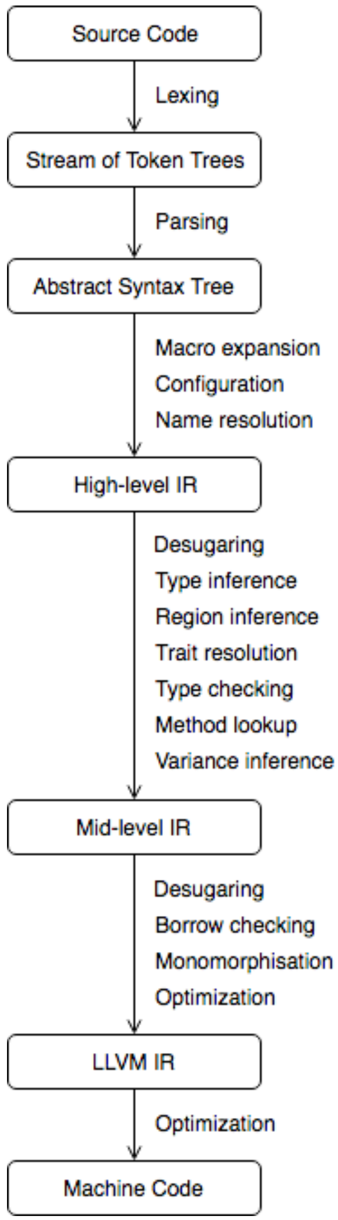

Figure 2: Overview of rustc.

can for instance specify that a function should only be compiled if the target operating system is Linux. In consequence, the AST node for the function declaration will be stripped out when compiling to other operating systems. Compilation can also be configured by supplying compiler flags, or through special comment annotations at the top of the source file, known as *header commands* [46, Ch. 4].

Name resolution Macro expansion and configuration is followed by name resolution [46, Ch. 12]. The AST is traversed in top-down order and every name encountered is resolved, i.e., linked to where it was first introduced. Names can be part of three different namespaces: values, types, or macros. The product of name resolution is a name-lookup index containing information about the namespaces. This index can be queried at later stages of compilation. In addition to building an index, name resolution checks for name clashes, unused imports, typo suggestions, missing trait imports, and more.

Transformation to HIR Upon finishing resolution and expansion, the AST is converted into a high-level IR (HIR) [46, Ch. 13]. The HIR is a desugared and more abstract version of the AST, which is more suitable for subsequent analyses such as type checking. For example, the AST may contain different kinds of loops, e.g., `loop`, `while` and `for`. The HIR instead represents all kinds of loops as the same `loop` node. In addition, the HIR also comes with a HIR Map which allows fast lookup of HIR nodes.

Type inference Rust's type inference algorithm is local to function bodies. It is based on the Hindley-Milner (HM) inference algorithm, with extensions for subtyping, region inference, and higher-ranked types [46, Ch. 15]. As input, the HM algorithm takes inference variables, also called existential variables, and unification constraints [52]. The constraints are represented as Herbrand term equalities. A Herbrand term is either a variable, constant or compound term. Compound terms contain subterms, and thus form a tree-like structure. Two terms are equated by binding variables to subterms such that their trees become syntactically equivalent. [53] Hence, the HM algorithm attempts to find a substitution for each inference variable to a type which satisfies the constraints. Type inference fails if no solution is found. Nominal types are equated by name and type parameters, and structural types by structure. Rust's inference variables are divided into two categories: type variables and region variables. Type variables can either be general and bind to any type, or restricted and bind to either integral or floating point types. Constraints for type variables are equality constraints and are unified progressively.

Region inference Region variables in contrast represent lifetimes for references [46, Ch. 15]. Constraints for region variables are subtype constraints, i.e., outlives relations. These are collected from lifetime annotations in item signatures and usage of references in the function body. Region inference is lazy, meaning all constraints for a function body need to be known before commencing the inference. A region variable is inferred as the lower-upper bound (LUB) of its constraints. The LUB corresponds to the smallest scope which still encompasses all uses of the reference. The idea is that a borrowed resource should be returned to its owner as soon as possible after its borrowers are finished using it. Lifetimes in Rust are currently lexical. Thereby, a lifetime, or region, is always bound to some lexical scope. This model will be changed in the near future to non-lexical lifetimes (NLL) which allow for more fine-grained control [42]. NLL are resolved through liveness analysis. Thus, a NLL ends when its value or reference is no longer live, i.e., when it will no longer be used at a later time. While it is possible to determine a lexical lifetime through the HIR, NLLs are derived from the MIR.

Trait resolution During trait resolution, references to traits are paired with their implementation [46, Ch. 16]. Generic functions can require parameters to implement a certain trait. The compiler must verify that callers to the function pass parameters that fulfill the obligation of implementing the trait. Trait implementations may as well require other traits to be implemented. Trait resolution fails either if an implementation is missing or if there are multiple implementations with equal precedence causing ambiguity.

Method lookup Method lookup involves pairing a method invocation with a method implementation [46, Ch. 17]. Methods can either be inherent or extensions. The former are those implemented directly for nominal types while the latter are implemented through traits, e.g., `impl Bar` and `impl Foo for Bar` respectively. When finding a matching method, the receiver object might need to be adjusted, i.e., referenced or dereferenced, or coerced to conform to the expected self-parameter.

Transformation to MIR After type checking finishes, the HIR is transformed into a heavily desugared Mid-Level Intermediate Representation (MIR). MIR resembles a control flow graph of basic blocks. Thus, there is no more nested structure, and all types are known to the compiler.

Borrow checking Borrow checking is what enforces Rust's ownership system. The borrow checker must ensure a number of properties, which include verifying that no variable is used before being initialized, that resources cannot have multiple owners, that resources are not moved while borrowed, and more. The algorithm takes the MIR, and previously inferred region variables, as input. Using dataflow analysis, the borrow checker will compute the owner for each resource by analysing how it is moved between variables. The regions and move data are thereafter used for checking the validity of each borrow.

Conclusively, the task of embedding Rust as a DSL in a host language will require the host language to support part of Rust's syntax and semantics. Hence, the host language must, to some degree, be able to impose the static checks of the rustc compiler. Optimally, the host language already has features in place, similar to Rust, which can be piggybacked on. While static syntactic checks and type inference are common, region inference and borrow checking are at large exclusive to Rust. Therefore, these might be more difficult to embed. The next chapter will showcase a subset of Scala's features that could be useful towards embedding Rust.

4 The Scala Programming Language

Scala is a high level object-oriented and functional programming language [14, Ch. 1]. It combines the core features of Java, e.g., static typing and generic collections, with a concise and elastic syntax. Because Scala is a JVM language, Scala and Java code integrate well with each other.

4.1 Basics

Classes, traits, and objects are the basis of Scala programs [55]. These are containers for members. A member is either a value, variable, method, type member, class, trait, or object. Values and variables store the result of an expression, and are immutable and mutable respectively. Methods are expressions which take parameters as input. Traits can contain abstract members and are equivalent to mixins in other languages. Classes in contrast require their members to be concrete, unless declared abstract. In addition, members can be included from multiple traits but can only be inherited from a single super class. A class or trait can also be declared as sealed, requiring all its subtypes to be declared within its file scope. Objects are singleton instances of classes.

Listing 8 Class, trait, and val.

```scala
abstract class Animal {
  def sound()
}
trait Fur {
  val color: String
}
class Bear(color: String) extends Animal with Fur {
  def sound() = println("Growl")
}

val grizzly = new Bear("brown")
grizzly.sound() // Growl
```

Case classes are classes whose members are by default immutable and public [55]. Methods, including `toString`, `equals`, `hashCode`, `apply`, and `unapply` are automatically generated for case classes. The `equals` method compares case classes by structure, rather than reference. The `apply` method is a shorthand constructor which can be invoked without needing to specify the `new` keyword. The `unapply` method allows case classes to be deconstructed for pattern matching. To this end, case classes serve as a convenient alternative to ordinary classes.

Listing 9 Case class.

```
case class Snake() extends Animal {
  def sound() = println("Hiss")
}

val cobra = Snake()
cobra.sound()
```

4.2 Implicits

One of Scala's special features are *implicits* [55]. A value or method can be declared as implicit, which permits the compiler to insert it automatically into parts of the program. There are two kinds of implicits: implicit parameters and implicit conversions.

An implicit parameter is a parameter to a method or class which does not need to be passed explicitly by the user. Instead, the compiler will search for an implicit of the specified type and insert it automatically. Listing 10 declares an implicit value `foo`, and a method `bar` with an implicit parameter. When called, `bar` requests implicit evidence for a value of type `Int`. In consequence, `foo` is inserted automatically as an argument into the method call.

Listing 10 Implicit parameters.

```
implicit val foo: Int = 3
def bar(implicit ev: Int): Int = ev
println(bar) // 3
// Desugards to println(bar(foo))
```

Implicit conversions allow the compiler to automatically convert an object into another object of a different type. In listing 11, method `m` is invoked on class `C`. Since `C` does not implement `m`, the compiler will search for an implicit conversion from `C` to a class implementing `m`. In this case, `conv` provides an implicit conversion from `C` to `I`. `I` implements `m`, and as a result `o` is converted to an object of class `I`.

Listing 11 Implicit conversions

```
class C
class I {
  def m: String = "Hello"
}
implicit def conv(from: C): I = new I
val o: C = new C
o.m // "Hello"
// Desugars to conv(o).m
```

Scala offers *implicit classes* which are syntactic sugar for extending classes with new members through implicit conversions. The code in listing 12 injects the method `m` into class `C`, and is equivalent to the code in listing 11.

Listing 12 Implicit classes

```
class C
implicit class I(o: C) {
  def m: String = "Hello"
}
val o: C = new C
o.m // "Hello"
```

4.2.1 Type Classes

Scala does not support type classes as first class citizens, but can express them through implicits. Listing 13 illustrates how `String` and `Int` can be extended with a type class `Print` to output values. The `Print` trait is a type class, and `printString` and `printInt` are instances of the type class. Scala provides special syntax for instantiating traits or abstract classes which contain a single abstract method (SAM) [55, Ch. 6]. Instances of a SAM can be written as a lambda function, rather than a method. For example, since `Print` contains a SAM, `printInt` can be written as a lambda function, which is less verbose than `printString`.

Listing 13 Type class

```
// Type class
trait Print[T] {
  def print(s: T): Unit
}
// Type class instance
implicit def printString: Print[String] = new Print[String] {
  def print(s: String) = println(s)
}
// Type class instance (SAM)
implicit def printInt: Print[Int] = v => println(v.toString)

def test[T](s: T)(implicit ev: Print[T]) = ev.print(s)
test(3)     // Int: 3
test("foo") // String: foo
```

4.2.2 Implicit type inference

Another use of implicit is to guide Scala's type inference. For instance, the code in listing listing 14 illustrates how implicits can be used to infer a default type for a type parameter `T` [56]. If `T` is unspecified when calling `test`, it will by default be inferred to `Int`. This is accomplished through two implicits. Implicit resolution will resolve to the most specific implicit, which is in this case `foo`. The difference between `foo` and `bar` is how `foo` requires both type parameters of `Default` to be of the same type. Hence implicit resolution for `Default[T,Int]` will resolve to `foo` if `T` is unbound, and thereby unify `T` with `Int`. In contrast, `Default[Double,Int]` will resolve to `bar` since the types do not match.

Listing 14 Inferring a default type parameter, from.

```
trait Default[A,B]
object Default {
  implicit def foo[T]   = new Default[T,T] {}
  implicit def bar[T,D] = new Default[T,D] {}
}

def test[T](implicit ev0: Default[T,Int], ev1: ClassTag[T]) = println(ev1)
test         // Int
test[Double] // Double
```

Note how a `ClassTag` is needed in listing 14 for printing the name of type `T`. An object's class can be accessed through `myObject.getClass`, but this gives no information about its type parameters. The reason is the JVM erases type parameter information at runtime. Hence, a `List[Int]` and a `List[Double]` both have the same type `List` at runtime. To evade type erasure, one can implicitly request a `ClassTag` or `TypeTag` from the compiler. `ClassTags` solely contain information about a runtime class, i.e., a `ClassTag[List[Int]]` only gives information about `List`. `TypeTags` contain information available at compile time, i.e., `TypeTag[List[Int]]` includes information about both `List` and `Int`.

4.3 Shapeless

Shapeless is a generic programming library for Scala. Its core abstraction is heterogeneous lists (HLists) [57]. An HList is a linked list where each element may have a unique type. Unlike regular Lists, HLists preserve type information of their elements, and unlike Tuples, HLists abstract over arity. Listing 15 illustrates how tuples cannot be prepended, but preserve type information of their elements. Lists can be prepended, but lose type information since all elements are coerced into the most common supertype. HLists can be prepended, while preserving type information, and as a result overcome the trade off between tuples and lists.

Listing 15 Tuple, List and Shapeless HList.

```
val tuple = (5, "abc", 'o')
val list  = List(5, "abc", 'o')
val hlist = 5 :: "abc" :: 'o' :: HNil

// Arity test - Prepend one element
val tuplePrepend = 4 :: tuple // ERROR, :: is not a method of Tuple
val listPrepend  = 4 :: list
val hlistPrepend = 4 :: hlist

// Polymorphism test - Extract head integer element
val tupleHead: Int = tuple._1
val listHead:  Int = list(0) // ERROR, type mismatch
val hlistHead: Int = list(0)
```

The `A :: B` syntax is syntactic sugar for `::[A,B]`. An HList is thereby composed of nested type constructors. Similarly to how the last node in a linked list may point to null, the last type constructor is terminated by `HNil`, e.g., `Int::Int::HNil`. Shapeless also provides natural transformations for HLists. It is for example possible to map over the elements of an HList and apply a different function to each element depending on its type, as shown in listing listing 16.

Listing 16 Polymorphic function

```
> object toInt extends Poly1 {
    implicit def caseInt = at[Int](x => x)
    implicit def caseString = at[String](x => x.length)
    implicit def caseChar = at[Char](x => 1)
  }

> val hlist2 = hlist map toInt
hlist2: Int :: Int :: Int :: HNil = 5 :: 3 :: 1 :: HNil
```

Operations on HLists are implemented as type-classes. An example is the `Comapped` type class. Shapeless' documentation describes `Comapped` as a *"type class witnessing that the result of stripping type constructor `F` off each element of HList `L` is `Out`"* [58]. Listing listing 17 shows how `Comapped[A,Option]` witnesses that each element in `A` is wrapped inside an `Option` type constructor. `Comapped.Aux[A,Option,B]` witnesses that the result of stripping the `Option` off each element of `A` is `B`.

Listing 17 Comapped example.

```
type A = Option[Int] :: Option[Char] :: HNil
type B = Int :: Char :: HNil

implicitly[Comapped[A,Option]]       // OK
implicitly[Comapped.Aux[A,Option,B]] // OK
```

The main parts of the implementation for `Comapped` are displayed in listing listing 18. The `Comapped` trait is the type class. It takes HList `L` and type constructor `F` as type parameters and produces a HList `Out` type member. The `Comapped` companion object declares an `Aux` type alias. `Aux` is a common pattern in type level programming for representing type members as type parameters [59, Ch. 3]. The convention is to unify the result of the type level computation with the last type parameter in `Aux`.

Next are the type class instances `hnilComapped` and `hlistComapped`. The former witnesses that the `Comapped` of an empty HList is `HNil`. The latter witnesses that the result of stripping `F` from the head `F[H]` of `L` and comapping the tail `T` to `TCM` is `H::TCM`. `Poly1` uses the same technique as `Comapped`, but lets the user add custom type class instances for handling elements of specific types.

Listing 18 Comapped implementation, from [@HlistOps].

```
trait Comapped[L <: HList, F[_]] {
  type Out <: HList
}

object Comapped {

  type Aux[L <: HList, F[_], Out0 <: HList] =
    Comapped[L, F] { type Out = Out0 }

  implicit def hlistComapped[H, T <: HList, F[_], TCM <: HList](
    implicit ev: Comapped.Aux[T, F, TCM]
  ): Aux[F[H]::T, F, H::TCM] =
    new Comapped[F[H]::T, F] { type Out = H::TCM }

  implicit def hnilComapped[F[_]]: Comapped[HNil, F] =
    new Comapped[HNil, F] { type Out = HNil }
}
```

Another feature of Shapeless are records [57]. A record is an `HList` where each value has an associated key. Keys are encoded as singleton typed literals and can be used to lookup or update values of the record. If a variable has a singleton type, then that variable can only assume one possible value. Singleton types have always been used internally by the Scala compiler, but there has not been syntax to express them externally. Shapeless exposes singleton typed literals to the programmer through implicit macros. In listing listing 19, `"foo" ->> 5` creates a value of type `FieldType[String("foo"), Int]`, which is an alias for `Int extends KeyTag[String("foo"), Int]` [60]. `String("foo")` is a singleton type. The call to `record("foo")` requests implicit evidence for the existence of type `FieldType[String("foo"),V]` in the record, where `V` is unbound. The evidence is as a result the associated value of type `V`.

Listing 19 Shapeless records.

```
> val record = ("foo" ->> 5) :: ("bar" ->> "abc") :: ("qux" ->> 'o') :: HNil
> val value = record("foo")
value: Int = 5
```

Future versions of Scala will bring first-class language support for singleton types, as depicted in listing listing 20 [61]. Another library which depends on singleton types is refined [62]. Refined adds refinement types to Scala which are types that constrain their values. For instance `Int Refined Positive` constrains the `Int` to only accept positive values.

Listing 20 Refined

```
val five: 5 = 5 // Ok
val four: 4 = 5 // Error

// refined
val foo: Int Refined Positive = 3     // Ok
val bar: Int Refined Positive = -2    // Error
val qux: Int Refined Greater[20] = 22 // Ok
val baz: Int Refined Greater[20] = 10 // Error
```

4.4 Scala DSLs

Scala's flexible syntax and semantics make it a suitable host language for embedding DSLs [14]. Most operators in Scala are implemented as methods, and can therefore be overloaded for declarativity. Furthermore, implementation details and boilerplate can be hidden from the user through implicits. Scala also provides facilities for compile time and runtime metaprogramming. During compile time, Scala's macro system can freely modify the Scala AST, which gives the programmer complete control over the semantics. Programs can further be reified to an AST, transformed and re-compiled at runtime using runtime-reflection. The following sections describe common patterns for embedding DSLs in Scala.

4.4.1 Fluent Interfaces and Method Chaining

A fluent interface is a technique for initializing objects through a chain of method calls [63, Ch. 3]. Each call instantiates a new object which is an updated version of the old object. As an example, listing 21 highlights a fluent interface '`Animal`' which is initialized and printed through a chain of method calls.

Listing 21 Fluent interface.

```
case class Animal(species: String = "", color: String = "") {
  def setSpecies(newSpecies: String) = Animal(newSpecies, color)
  def setColor(newAge: Int)          = Animal(species, newColor)
  def print                          = println(color + " " + species)
}
Animal()
  .setSpecies("Fox")
  .setColor("Gray")
  .print() // Gray Fox
```

Through representing assignments as method calls, the object's fields can be updated without being declared as mutable. By also chaining methods one after the other, the user does not need to declare excess local variables for storing intermediate results. Spark's RDD API and Flink's DataStream API are instances where this technique sees use [6][7].

4.4.2 String literals

A crude approach to embedding a DSL in Scala is to envelop its code in a string literal, as shown in listing 22. Although this approach is easy to program, Scala is unable to statically check check the embedded code. An example where this pattern sees use it OpenCL, where source code for the GPU can be written inside raw C string literals [64]. Scala's string literals can either be written as single-line or multi-line [14]. The latter is effective for language embedding because it is a continuous block of text with fewer escape characters, since double-quotes need not be escaped.

Listing 22 String literal.

```
val snippet =
  """
    object Test {
      println("Hello world")
    }
  """
```

4.4.3 Language Injection

Language injection is a feature offered by the IntelliJ IDEA editor for metaprogramming [65]. It can be used to imbue syntax highlighting and linting for a programming language, e.g., HTML, CSS, SQL, RegExp, into a string literal. The feature can either be enabled temporarily through IntelliJ's interface, or by placing comment annotations in the source code as in listing 23. The IntelliJ-Rust plugin for IntelliJ adds support for Rust Language Injection [66], and is capable of statically checking code up until name resolution. It will therefore detect errors such as name clashes but will miss type mismatches and ownership violations.

Listing 23 Language Injection.

```
// language=Scala
val lintedSnippet =
  """
    object Test {
      println("Hello world")
    }
  """
```

4.4.4 String Interpolation and Quasi Quotation

String interpolation is a feature in Scala which lets the user annotate that rules should be applied to a string literal [14, p. 153]. In listing 24, the `s` string interpolator splices, i.e., inserts, an external value into the string literal. Scala desugars the string literal into a `StringContext`, and invokes the string interpolation method on it. External values are passed as arguments to the method call. Custom string interpolation methods can be implemented for the `StringContext` through type classes

Listing 24 The 's' string interpolator

```
val x = "world"
s"Hello $x." // Desugars to: new StringContext("Hello ", ".").s(x)
```

Scala has a collection of advanced string interpolators for metaprogramming, referred to as quasi-quotes [67]. A quasi-quote takes a string literal containing a snippet of Scala code and constructs a corresponding Scala AST. Quasi quotes may be spliced with type names, term names, modifiers, flags, symbols, ASTs, and more. This facilitates the possibility of merging quasi quotes together to form a larger AST. ASTs can consequently be compiled down to byte code with runtime reflection, or deconstructed with pattern matching. Spark SQL's Catalyst optimizer relies on quasi quotes for translating incoming SQL queries into Scala byte code [17]. Scala's native quasi quotes are not type safe, and are limited to only supporting Scala code. Support for another language could be added by writing or generating a lexer and parser. Alternatively, Squid is an open source project for creating custom type safe Scala quasi-quote DSLs [68].

Listing 25 Scala quasi-quotes

```
val abstractSyntaxTree =
  q"""
    object Test {
      println("Hello world")
    }
  """
```

4.4.5 Algebraic Data Types

Algebraic Data Types (ADTs) are a category of recursive data types with algebraic properties [14, Ch. 16]. Algebras are defined by objects, operations, and laws. Objects can for example be numbers or booleans. Operations combine objects into new ones. Laws dictate the relationships between objects and operations, e.g., associative, commutative, and distributive laws. As an example, a linked list is an ADT. Scala's `List` has two subtypes: `Nil` and `::`. The former is the empty `List`, and the latter combines an element with a `List` to form a new `List`. The number of values a `List` can attain is thus confined, and make it possible to reason clearly about the `List`'s behavior. ADTs are suited for embedding DSLs in abstract syntax. For example, listing 26 defines an algebra of integer literals and addition which is subsequently interpreted with `eval` [69].

Listing 26 ADT example [@GADT].

```
sealed trait Exp
case class Lit(v: Int)         extends Exp
case class Add(l: Exp, r: Exp) extends Exp

def eval(exp: Exp): Int = exp match {
  case Lit(v)   => v
  case Add(l,r) => eval(l)+eval(r)
}

eval(Add(Lit(1), Add(Lit(9), Lit(3)))) // 13
```

4.4.6 Generalized Algebraic Data Types

An issue with the ADT in listing 26 is the expectation that every expression evaluates to `Int`. This prevents the ADT from being extended with new types of algebra, e.g., `Boolean` algebra. One solution is to couple each expression with some form of evidence, i.e., type tag, indicating what it evaluates to. This kind of encoding type checks at runtime, and is known as a *tagged* encoding [70]. Generalized Abstract Data Types (GADTs) generalize parameterized ADTs over the types expressions evaluate to [71]. Hence, GADTs have *tagless* encoding since no type tags are involved. The DSL in listing 27 displays how listing 26 can be generalized and extended with the `Eq` operator of `Boolean` algebra [69].

Listing 27 GADT DSL example

```scala
sealed trait Exp[T]
case class Lit[T](v: T)                      extends Exp[T]
case class Add(l: Exp[Int], r: Exp[Int]) extends Exp[Int]
case class Eq(l: Exp[Int], r: Exp[Int])  extends Exp[Boolean]

def eval[T](exp: Exp[T]): T = exp match {
  case Lit(v)    => v
  case Add(l, r) => eval(l) + eval(r)
  case Eq(l, r)  => eval(l) == eval(r)
}
eval(Eq(Add(Lit(1), Lit(2)), Lit(3))) // true
```

4.4.7 Tagless Final

ADTs and GADTs both suffer from the expression problem [70]. The expression problem is defined as:

> "The expression problem is a new name for an old problem. The goal is to define a datatype by cases, where one can add new cases to the datatype and new functions over the datatype, without recompiling existing code, and while retaining static type safety (e.g., no casts)." - Philip Wadler [72]

Listing 26 and listing 27 modularize by interpretations and as a result are difficult to extend with new algebras. Tagless Final is an embedding which solves the expression problem [70]. While ADTs and GADTs build a tree of values, Tagless Final instead forms a tree of function calls. The former two have an *initial* encoding, and Tagless Final, as the name denotes, has a *final* encoding. Tagless Final solves the expression problem through type classes. Listing 28 re-implements the ADT in listing 26 with a tagless final encoding. `Lit` and `Add` are type classes defining the syntax of the language. The `evalLit`, `evalAdd`, `showLit` and `showAdd` implicits are type class instances defining the DSL's semantics. The first two evaluate to an `Int`, and the second two concatenate a `String`. The `test` method implicitly imports the semantics into scope. By de-coupling constructs from interpretations, the program becomes easier to extend. Creating a new construct involves defining a new type class, along with type class instances for it covering all interpretations. Creating a new interpretation is the opposite.

Listing 28 Tagless Final example [@TaglessFinalScala]

```
trait Lit[T] { def apply(i: Int): T     }
trait Add[T] { def apply(l: T, r: T): T }

implicit val evalAdd: Add[Int] = (l, r) => l + r
implicit val evalLit: Lit[Int] = i => i

implicit val showAdd: Add[String] = (l, r) => s"($l + $r)"
implicit val showLit: Lit[String] = i => i.toString

def test[T](implicit add: Add[T], lit: Lit[T]): T =
  add(add(lit(1), lit(9)), lit(5))

test[Int]    // 15
test[String] // ((1 + 9) + 5)
```

5 Design

This section describes the design of the two DSLs which were developed.

5.1 Shallow Embedded DSL

The first DSL is a basic string interpolator which splices snippets of Rust code together. Snippets are statically checked by passing them to the rustc compiler or the syn parser, depending on what is specified by the user. The rustc compiler can only check snippets which represent whole compilation units. In contrast, the syn parser can check individual constructs, but only syntactically. Both a runtime and compile time version of each string interpolator was implemented. The compile time version can only splice snippets in the form of literals, known at compile time.

5.2 Deeply Embedded DSL

The second DSL is statically typed with a deep embedding. It is designed to be extensible in the direction of defining custom structs, functions, and traits. Meanwhile, the implementation of constructs and interpretations is abstracted away from the user. The DSL's API is a combination of fluent interfaces for defining items, and parameterized ADTs for constructing an AST. The Shapeless library is used to make the fluent-interfaces polymorphic and type safe. By building an AST, the DSL has a deep embedding which supports multiple interpretations. The only interpretation available at the time being is Rust code generation.

5.2.1 Overview

Listing 29 illustrates the directory-structure of the deeply embedded DSL. There are three modules: constructs, types, and interpretations.

Listing 29 Directory tree of the project.

```
Rust-DSL/
+-AST.scala
+-Constructs/
| +-File.scala
| +-Expressions/
| | +-Literal.scala
| | +-Operators.scala
| | +-ControlFlow.scala
| | +-Let.scala
| +-Items/
| | +-Function.scala
| | +-Struct.scala
| | +-Trait.scala
| | +-Impl.scala
| +-Verbatim.scala
+-Types.scala
+-Interpretations/
  +-Showable.scala
```

AST defines the node and interpretation categories of the DSL. Constructs define parameterized ADT-encoded nodes for building the AST. File is the root-node of the AST. Expressions contain nodes for representing literals, control-flow expressions, operators, and let-declarations. Items provide fluent interfaces for building functions, structs, traits, and trait implementations. The main missing constructs are enums, tuples, arrays, macros, type aliases, and patterns. At the moment, these can be written in Verbatim if necessary. Types encompass Rust's primitive, reference, and nominal types. It also contains utilities for guiding Scala's type inference. Interpretations provide a `Showable` interpretation which nodes extend to generate code. Types in contrast use a type class for generating code. A program is synthesized by traversing its AST in top-down order. By default, types in the generated code have the same identifier as in the DSL. Identifiers for functions, variables, fields, and arguments, are specified by the user.

5.3 Emulating `rustc`

The DSL attempts to issue a subset of `rustc`'s static checks through `scalac`. Tbl. 2 lists supported and non-supported checks.

Table 2: Static checks of the deeply embedded DSL.

Static checks	Supported
Syntactic checks	Yes
Macro expansion	No
Name resolution	Yes
Type inference	Yes
Type checking	Yes
Lifetime elision	No
Region inference	No
Trait resolution	Yes
Method lookup	Yes
Coersions	No
Mutability checking	No
Borrow checking	No

ADTs ensure grammatical correctness since each node requires its children to be a specific syntactic unit. For example, a binary addition operator requires its two operands to be expressions, and will reject anything else. ADT parameterization also establishes type safety, as both operands are then required to evaluate to the same type. If a type parameter is unspecified, Scala will attempt to infer it. Programs are written in higher-order abstract syntax for automatic name resolution. In other words, when a node refers to a variable, it will refer directly to the node of the variable rather than its identifier. Traits in Rust are programmed as type classes in Scala. Thereby, trait resolution piggybacks on implicit resolution. The main unsupported semantics are mutability checks, coercions, lifetime inference and borrow checking.

6 Implementation

The implementation is in pure Scala with one dependency to the Shapeless library. Usage of macros is sparse.

6.1 Shallow Embedded DSL

The shallow embedded DSL is a runtime and compile time string interpolator, as shown in listing 30. The interpolators are implemented as extension methods and extension macros for the `StringContext` class respectively. Both take an option which indicates how the spliced string should be statically checked. Four options are available: `"item"`, `"expr"`, `"type"`, and `"file"`. The first three tell the interpolator to verify that the string is a grammatically sound item, expression, or type, by piping it through Rust's `syn` parser. In contrast, `"file"` saves the snippet to a temporary file on disk and compiles it with `rustc`.

Listing 30 Rust string interpolators.

```
implicit class RustInterpolator(val sc: StringContext) {
  def rrust(args: String*)(option: String): String = rustrImpl(sc, opt, args)
  def crust(args: String*)(option: String): String = macro rustcImpl
}
```

Listing 31 contains the runtime interpolator method. The call to `getSnippet` splices the string parts with the arguments into a snippet. Next, `checkSnippet` passes the snippet to `rustc` or `syn` in a separate process, depending on the option. If the process reports an error, i.e., the error code is not zero, a runtime exception is thrown with the snippet and error message, otherwise the snippet is returned.

Listing 31 Runtime string interpolator method.

```
def rustrImpl(sc: StringContext, option: String, args: Seq[String]): String = {
  val snippet = getSnippet(sc.parts, args)
  val (errorCode, errorMsg) = checkSnippet(snippet, option)
  if(errorCode != 0) {
    throw new RustException(s"$errorMsg\n>> $snippet <<\n")
  }
  snippet
}
```

Listing 32 defines the compile time interpolator macro. The logic is at large the same as the runtime interpolator, with a difference being how the parameters need to be excavated from the AST. For example, `option` is extracted from `expOption.tree` through `Literal(Constant(option: String))`. As a requirement, all `expArgs` must be string literals. If `rustc` or `syn` reports an error, the macro will abort compilation with an error message.

Listing 32 Compile time string interpolator macro.

```
def rustcImpl(c: Context)(expArgs: c.Expr[String]*)
                         (expOption: c.Expr[String]): c.Expr[String] = {
  import c.universe._
  val args = expArgs.map(expArg => getSplice(c)(expArg))
  val parts = getParts(c)
  val Literal(Constant(option: String)) = expOption.tree
  val snippet = getSnippet(parts, args)
  val (errorCode, errorMsg) = checkSnippet(snippet, option)
  if(errorCode != 0) {
    c.abort(c.enclosingPosition, s"$errorMsg\n>> $snippet <<\n")
  }
  c.Expr[String](Literal(Constant(snippet)))
}
```

6.2 Deeply Embedded DSL

The construct and interpretation categories for the deeply embedded DSL are illustrated in listing 33. Constructs are required to implement the `show` method from the `Showable` trait. The `show` method generates code by directly printing it to a file with a designated `PrintWriter`. `ShowName` and `Show` are type classes for showing types. Place expressions are distinct in how they can both be evaluated and assigned to. Only a subset of items can be declared in traits. For example, a trait may contain a method but not a struct. `Items` which can appear in traits are `TraitItems`. Expressions, let- and item declarations can all occur in statement position.

Listing 33 Interpretations and constructs.

```
// Interpretations
trait Showable { def show(pw: PrintWriter): Unit }
trait ShowName[T] extends Showable // Type class
trait Show[T]     extends Showable // Type class
// Constructs
trait Node        extends Showable // AST Node
trait Pat         extends Node     // Pattern
trait Stm         extends Node     // Statement
trait Exp[T]      extends Stm      // Expression
trait PlaceExp[T] extends Exp[T]   // Place Expression
trait Item        extends Stm      // Item Declaration
trait TraitItem   extends Item     // Trait Item Declaration
```

Constructs are implemented as SAM instances wherever possible. As an example, the root of the AST is a `File` node, in listing 34, taking a variable number of item declarations. Code for `File` is generated by invoking `show` on its item declarations. For convenience, an implicit class with a method `sep` is implemented for showing delimited sequences of `Showable` objects.

Listing 34 File implementation.

```
def File(items: Item*): Node = pw => items.sep("\n").show(pw)
```

6.2.1 Types

The DSL represents proper types directly as proper Scala types. Polymorphic types are depicted as Scala types which extend a `Polymorphic` type, defined in listing 35. The generics of the polymorphic type are stored inside an `HList` type parameter. Moreover, references are represented as `ref[L,T]` where `L` is a lifetime and `T` is a type. `A`, `B`, and `C` in listing 35 are examples of proper types, polymorphic types and references respectively. Tuples, arrays, and slices are not currently supported.

Listing 35 Types

```
trait Polymorphic { type Generics <: HList }
trait ref[L,T]
// Example
trait i32; trait bool; trait static
trait HashMap[K,V] extends Polymorphic { type Generics = K::V::HNil }
type A = i32                 // i32
type B = HashMap[i32,bool] // HashMap<i32,bool>
type C = ref[static,i32]    // &'static i32
```

Type and lifetime annotations are a prerequisite for item declarations. Given an AST node parameterized by a type `T`, it must be possible to show `T`. The DSL uses a type class `Show[T]` to show a type `T`. By default, implicit resolution for `Show[HashMap[i32,bool]]` will produce `"HashMap<i32,bool>"`. A second type class `ShowName[T]` exists for only showing the name of a type, e.g., `ShowName[HashMap[i32,bool]]` outputs `"HashMap"`. By having two type classes, types can be printed differently depending on the context. In listing 36, the type class instance `showName` extracts the name of a type from a `ClassTag` and prints it. The `showProper` and `showPoly` instances show proper types and polymorphic types respectively. There is no ambiguity between the two as the latter has a more specific return type.

Listing 36 Type class instances for showing types.

```
implicit def showName[T](implicit ev: ClassTag[T]): ShowName[T] =
  pw => pw.print(ev.toString.split("(\\.|\\$)").last)

implicit def showProper[T](implicit ev: ShowName[T]): Show[T] =
  pw => ev.show(pw)

implicit def showPolymorphic[T <: Polymorphic, L <: HList]
(implicit ev0: ShowName[T],
          ev1: LiftAll.Aux[Show, T#Generics, L],
          ev2: ToTraversable.Aux[L, List, Show[_]]
): Show[T] = pw => {
    val list = ev1.instances.toList[Show[_]]
    show"$ev0<${list.sep(",")}>"(pw)
  }

// Example
implicitly[Show[HashMap[i32,bool]]].show(pw) // HashMap<i32,bool>
```

A type `Show[HashMap[i32,bool]]` will resolve to `showPolyType` which in effect requests evidence for `ShowName[HashMap[_]]` to extract the name `"HashMap"`. Afterwards, Shapeless' `LiftAll` retrieves the `Show` type class instance for each element, `i32::bool::HNil`, in the list of generics [57]. Finally, Shapeless' `ToList` transforms the result of `LiftAll` into a `List[Show[_]]`.

A string interpolator `show` is implemented for showing a string, spliced with object of type `Showable`. For example, `show"<$x>"(pw)` will execute `{pw.print("<"); x.show(pw); pw.print(">")}`. Furthermore, an implicit conversion from `String` to `Showable` exists, allowing `Strings` to be spliced as well.

It is possible to override what a type generates to by assigning it a custom type class instance of `ShowName`, as displayed in listing 37. For example, Rust's unit type `()` can be expressed by overriding `unit`.

Listing 37 Overriding types.

```
trait unit
implicit val showunit:    ShowName[unit]       = () => "()"
implicit val showHashMap: ShowName[HashMap[_]] = () => "BinaryTree"
// Example
implicitly[Show[HashMap[i32,unit]]].show(pw) // BinaryTree<i32,()>
```

6.2.2 Literals

Every expression is parameterized by an invariant type parameter, indicating what the expression evaluates to. One class of expression are literals, defined in listing 38. The value a literal evaluates to is stored as a string. Values of primitive types in Rust could alternatively be stored by their equivalent type of value in Scala. For example, the value of an `f32` in Rust could be stored as an `Float` in Scala. This could however potentially result in unexpected round-off errors if Scala decides to treat floats differently than Rust. By using strings, the code for literals is guaranteed to be generated exactly as specified by the user.

Listing 38 Add

```
case class Lit[T](value: String): Exp[T] = pw => pw.print(value)
// Example
val i = Lit[i32]("1")
val b = Lit[bool]("true")
i.show(pw) // 1
b.show(pw) // true
```

6.2.3 Unary Operators

Rust's unary and binary operators are listed in tbl. 3. Scala is able to overload the unary prefix operators '+', '-', '!' and '~' [73]. Other unary prefix operators can be programmed as regular methods, and as a result require parentheses at the use-site.

Table 3: Rust's operators [74].

Category	Symbols
Unary	`-` `!` `&` `*`
Binary-Arithmetic	`+` `-` `*` `/` `%` `+=` `-=` `*=` `/=` `%=`
Binary-Relational	`<` `>` `<=` `>=` `==` `!=`
Binary-Logical	`&&` `||`
Binary-Bitwise	`^` `&` `|` `<<` `>>` `^=` `&=` `<<=` `>>=`
Other	`;` `=` `_` `:` `.` `..` `...` `@` `?`

Listing 39 defines the unary prefix operators '&', '*' and '!', i.e., reference-of, dereference, and logic negation. The '*' and '&' operators are regular methods which correspondingly unwrap and wrap a `ref[L,T]` around a type. When dereferencing, the result is a place expression. The '!' operator is overloaded for boolean expressions with an extension method `unary_!`.

Listing 39 Unary operators.

```
def &[T](v: Exp[T]): Exp[ref[static,T]] = pw => show"(&$v)"(pw)
def *[T](v: Exp[ref[_,T]]): PlaceExp[T] = pw => show"(*$v)"(pw)
implicit class Logical(v: Exp[bool]) {
  def unary_!: Exp[bool]                = pw => show"(!$v)"(pw)
}
// Example
val val1 = *(&(i))
val val2 = !b // Desugars to b.unary_!
val1.show(pw) // *(&(1)))
val2.show(pw) // !true
```

6.2.4 Binary Operators

Binary operators are programmed as extension methods as well. Listing 40 defines two operators, ‘`:=`’ and ‘`#:`’. The former assigns a value to a place expression, requiring both operands to be of the same type. The latter terminates a statement followed by an expression. Both end with a colon, and are in consequence right-associative. Right-associativity for ‘`=:`’ implicates `a =: b =: c;` is evaluated as `a =: (b =: c)`, as opposed to `(a =: b) =: c`. All assignment operators are right-associative, and certain operators, such as relational operators, require parentheses, e.g., `(x == y) == z`. The remaining operators are left-associative.

Listing 40 Binary operators for assignment and statement termination.

```
implicit class Assign[T](val r: PlaceExp[T]) {
  def =:(l: Exp[T]): Exp[unit] = pw => show"$l = $r"(pw)
}
implicit class Terminate[T](val r: Exp[T]) {
  def #:(l: Stm): Exp[T]       = pw => show"$l; $r"(pw)
}
// Example
val exps = (val1 =: i) #: b
exps.show(pw) // (*(&1)) = 1; true
```

6.2.5 If-Else

Listing 41 defines the `If` expression which requires a condition and if-clause. `If` can be chained with `Else` into an if-else, given that the if- and else-clause evaluate to values of the same type. Because `If` has more than one method, it cannot be written as a SAM instance.

Listing 41 If-Else definition.

```
def If[T](cond: Exp[bool])(ic: Exp[T]) = new Exp[T] {
  def Else(ec: Exp[T]): Exp[T] = pw => show"if $cond { $ic } else { $ec }"(pw)
  def show(pw: PrintWriter) =         show"if $cond { $ic }"(pw)
}
// Example
val if_else = If(b) { i+i } Else { i }
if_else.show(pw) // if true { 1+1 } else { 1 }
```

6.2.6 Let

A variable is treated by the DSL as a function with one argument. For example, `{let x: i32 = 3; x+x}` is isomorphic to `(|x: i32| x+x )(3)`. The `Let` node in listing 42, for generating a `let` declaration, takes a `value`, an `id`, and a function `fn`. The `id` is by default generated with `new_id` which is a global counter, but can also be specified manually. The `value` is bound to a `variable` with name `id`. `Let` evaluates to the `result` of applying `fn` to the `variable`.

Listing 42 Let definition.

```
def Let[T,O](value: Exp[T], id: String = new_id)
            (fn: PlaceExp[T] => Exp[O]): Exp[O] =
 pw => {
    val variable: PlaceExp[T] = pw => pw.print(id)
    val result = fn(variable)
    show"{let $variable = $rhs; $result}"(pw)
 }
// Example
val let1 = Let(i) { x => x+x }
let1.show(pw) // {let x1 = 1; (x1 + x1)}
val let2 = Let(i, "x") { x => x+x }
let2.show(pw) // {let x = 1; (x + x)}
```

6.2.7 Functions

Functions, and other items, are implemented with fluent interfaces. The idea is to construct the function incrementally through method chaining, adding arguments, bounds, attributes, etc. When finished, the function can either be declared, called or inlined. The parameters of the fluent interface are declared in listing 43. A function must have a name, but the rest, i.e., attributes, bounds, arguments, and body, is optional. Arguments are stored in both a Shapeless `HList` and a regular `List`. The purpose of the `HList` is to abstract over the arity of the input argument list, and ensure type safety inside the function body. The `List` is solely used for showing the arguments. The `FnArg` type can either be instantiated as an ordinary argument or a self-argument. By default, the return type of a function is the unit type `()`

unless specified otherwise, e.g., `fn main() { ... }` desugars to `fn main() -> () { ... }`. Hence, `O`, which is the return type, is by default inferred to `unit`, using the `Default` type class described in the background.

Listing 43 Fluent interface parameters (Function)

```
case class Fn[I <: HList, O:Df[unit]#L](
  NAME:    String              = "",
  ATTRS:   String              = "",
  BOUNDS:  String              = "",
  ARGS:    I                   = HNil,
  ARGLIST: List[FnArg[_]]      = List(),
  BODY:    Option[I => Exp[O]] = None
) extends Exp[O] {
```

Listing 44 describes the implementation for ordinary arguments. An argument may either appear declared in a function signature, inserted at a call-site, used in a function body, or inlined as a local variable. Arguments must therefore be able to show their identifier, expression, and type. The identifier is given using Shapeless' `Witness.Aux[K]` which extracts the value from a singleton type literal.

Listing 44 Argument definition

```
def newArg[K,V](exp: Exp[V])
              (implicit ev1: Witness.Aux[K], ev2: Show[V]) = new FnArg[V] {
  val id = ev1.value.toString
  def decl:   Stm = pw =>     show"$id: $ev2"(pw)
  def inline: Stm = pw =>     show"let $id = $exp"(pw)
  def insert: Stm = pw =>     show"$exp"(pw)
  def show(pw: PrintWriter) = show"$id"(pw)
}
```

Each parameter in the interface comes with setter method. For instance, `arg` in listing 45 is a method part of the fluent interface for adding an argument to the function. As input, `arg` expects an record field where the value is an expression, e.g., `("id" ->> exp)`. A corresponding field is then created and added to the fluent interface. `K` and `V` require view bounds for propagating the implicits the `newArg` method expects.

Listing 45 Staging a function argument

```
def arg[K:Witness.Aux, V:Show](exp: FieldType[K, Exp[V]]) = {
  val arg = field[K](newArg[K,V](exp))
  Fn(NAME, ATTRS, BOUNDS, arg::ARGS, arg::ARGLIST, None)
}
```

Listing 46 displays how the fluent interface declares a method. Since Rust demands explicit type annotations in the function signature, implicit evidence is needed for declaring the output type. Conditionally, the function body is only declared if it has been specified.

Listing 46 Unstaging a function

```
def decl(implicit ev: Show[O]): TraitItem = pw => {
  val input = ARGLIST.reverse.map(_.decl).sep(", ")
  if (BODY.isEmpty) {
    show"fn $NAME$BOUNDS($input) -> $ev;"(pw)
  } else {
    val output = BODY.get.apply(ARGS)
    show"fn $NAME$BOUNDS($input) -> $ev {$output}"(pw)
  }
}
```

The fluent interface can then be used as in listing listing 47 for writing a function which calculates the maximum value of two integers. `PH`, or PlaceHolder, is an object which can which can be implicitly converted into any other expression. In other words, `PH` can always be passed in place of an expression, and can be used as a placeholder for expressions when creating item declarations.

Listing 47 Function example

```
def max(x: Exp[i32], y: Exp[i32]) =
  Fn()
    .name("max")
    .arg("x" ->> x)
    .arg("y" ->> y)
    .returns[i32]
    .body { args =>
      If(args("x") > args("y")) { args("x") } Else { args("y") }
    }

max(i, i+i).show(pw)        // max(1, 1+1)
max(i, i+i).inline.show(pw) // if (1 > 1+1) { 1 } else { 1+1 }
max(PH, PH).decl.show(pw)
// fn max(x: i32, y: i32) -> i32 {if (x > y) { x } else { y }}
```

6.3 Structs

The fluent interface for structs employ the same strategy as functions. A struct has a type, and optionally fields, bounds, and attributes. These are defined in listing 48. `S` is the type of the struct, and `F` is the types of the fields. The struct's type is thereby separated from its fields, as is specified for nominal types.

Listing 48 Fluent interface parameters - Struct

```
case class Struct[S, F <: HList](
  NAME:      Showable          = "",
  BOUNDS:    String            = "",
  ATTRS:     String            = "",
  FIELDLIST: List[StructField] = List()
) extends Exp[S] { ... }
```

Listing 49 implements the '`f`' operator, equivalent to '`.`', for accessing fields. When invoking '`f`' on an expression `Exp[S]`, the compiler will attempt to find implicit evidence for `Struct[S,F]`, which if found unifies `F`. As input, the '`f`' operator takes a key and uses Shapeless' `Selector.Aux[F,key.T,V]` to unify the type `V` of its associated value. The product is a place expression of type `V`, representing the accessed field.

Listing 49 Struct field access

```
implicit class FieldAccess[S,F<:HList](exp: Exp[S])
                                      (implicit ev: Struct[S,F]) {
  def f[V](key: Witness)
          (implicit ev: Selector.Aux[F,key.T,V]): PlaceExp[V] =
    pw => show"$exp.${key.value.toString}"(pw)
}
```

As an example, listing listing 50 displays how to declare a `Point` struct with two integer fields. The implicit value is required as evidence for looking up the fields.

Listing 50 Struct example

```
trait Point
val point(x: Exp[i32], y: Exp[i32]) =
  Struct()
    .name[Point]
    .field("x" ->> x)
    .field("y" ->> y)
implicit val pointDefinition = point(PH,PH)

point(PH, PH).decl.show(pw) // struct Point { x: i32, y: i32 }
Let(point(i, i)) { s =>
  s.f("y")
}.show(pw) // let x1 = Point { x: 1, y: 1 }; x1.y
```

6.4 Traits

Traits and implementations in Rust are equivalent to type classes and type class instances. Given a trait `T` and a type `S`, it should be possible to request evidence for an implementation of `T` for `S`. Moreover, traits contain trait items which are either constant values, methods, type aliases, or macros [75]. At the moment, only methods are supported. Listing 51 illustrates how to declare a Rust trait `Math`, as a Scala class, with a method `square` that calculates the square product. To declare the trait, an instance of `Math` is passed to a fluent interface, which declares its members. The implementation for the fluent interface follows the same path as structs. A difference is how the fields of the struct are replaced by methods in the trait.

Listing 51 Trait example

```
trait Self
class Math {
  type S
  def square(self: Exp[S]) =
    Fn()
      .self("self" ->> self)
      .returns[i32]
}
def Math =
  Trait()
    .inst(new Math { type S = Self})
    .member(_.square(PH).decl)
math.show(pw)
// trait Math { fn square(self) -> i32; }
```

`Math` can subsequently be implemented for a specific type by instantiating it and overriding each method. By declaring the implementation as implicit, other methods can request implicit evidence for it.

7 Evaluation

The evaluation is in the form of a demonstration, testing the features and static checking capabilities of the two DSLs.

7.1 Shallow Embedded DSL - Validation testing

The runtime and compile time interpolators are evaluated with validation testing. Code for the tests is displayed in listing listing 52. Each test applies an interpolator `X` to a string, which splices it with arguments and then checks it with option `Y`. `X` and `Y` are the control variables. `X` is substituted with either `rrust` or `crust`, and `Y` is replaced by either `"file"` or `"item"`. Hence, each test is run with four variations. The tests can be described as:

- **Test 1**: Interpolate a string by splicing it with a string literal, into a statically correct string, and then check it. The result is asserted.
- **Test 2**: Interpolate a string by splicing it with a string literal, into a syntactically incorrect string, and then check it.
- **Test 3**: Interpolate a string by splicing it with a string literal, into an statically incorrect, but syntactically correct string, and then check it.
- **Test 4**: Interpolate a string by splicing it with the result of a compile time interpolator, into a statically correct string, and then check it.

Listing 52 Validation testing of Shallow DSL

```
// Test 1
val a = "i32"
assert(X"fn main() { let x: $a = 3; }"(Y) == "fn main() { let x: i32 = 3; }")
// Test 2
val b = ";"
X"fn main() { let x: $b = 3; }"(Y)
// Test 3
val c = "f32"
X"fn main() { let x: $c = 3; }"(Y)
// Test 4
val d = crust"(1+2)"("expr")
X"fn main() { let x = $d; }"(Y)
```

The results of the tests are listed in tbl. 4. True positive (`TP`) means the interpolator reported an error when it should have. False positive (`FP`) means the interpolator reported an error when it should not have. True negative (`TN`) means the interpolator did not report an error when it should not have. False negative (`FN`) means the interpolator did not report an error when it should have. In short, `TP` and `TN` denotes wanted behavior, and **`FN`** and **`FP`** denotes unwanted behavior. In the first test, no interpolator reported an error, as expected. In the second test, all interpolators were able to detect the syntax error. In the fourth test, only the interpolators with the rustc option were able to detect the type error. In the final test, only the runtime interpolators were able to splice the non-string literal input argument.

Table 4: Results from the validation test.

Test	rrust+rustc	rrust+syn	crust+rustc	rrust+syn
1	TN	TN	TN	TN
2	TP	TP	TP	TP
3	TP	**FN**	TP	**FN**
4	TN	TN	**FP**	**FP**

7.2 Deeply Embedded DSL - Demo

The deeply embedded DSL is evaluated by writing a program which combines the features described in the implementation. The demo implements a `Geometry` trait with a `perimeter` method for a generic `Rectangle`, similar to the example given in the Rust background. First, types and literals are declared in listing listing 53. '`A`' acts as a type which can later be printed as a generic.

Listing 53 Demo - Types and literals.

```
trait A
val i = Lit[i32]("5")
```

Then, listing 54 declares a struct `Rectangle` with a generic width and height along with an implicit lookup function. The bounds are specified as a string.

Listing 54 Demo - Struct

```
trait Rectangle[T] extends Polymorphic { type G = T::HNil }
def Rectangle[T:Show](w: Exp[T], h: Exp[T]) =
  Struct()
    .name[Rectangle[T]]
    .bounds("<A>")
    .field("w" ->> w)
    .field("h" ->> h)
implicit def RectangleDefinition[T:Show] = Rectangle[T](PH, PH)
```

Listing 55 declares a `Geometry` trait with a `perimeter` function. The trait must also have an implicit lookup method, and declaration method.

Listing 55 Demo - Trait with method

```
class Geometry[T:Show] extends Polymorphic {
  type G = T::HNil
  type S
  def perimeter(self: Exp[S]) =
    Fn()
      .name("perimeter")
      .self("self" ->> self)
      .returns[T]
}
implicit def GeometryDefinition = new Geometry[A]{type S = A}
def GeometryDeclaration =
  Trait()
    .bounds("<A>")
    .inst(new Geometry[A] { type S = Self})
    .member(_.perimeter(PH).decl)
```

Listing 56 implements the `Geometry` trait for the `Rectangle` struct, and also defines a declaration method. The perimeter method adds together the sides of the rectangle by accessing its fields.

Listing 56 Demo - Trait implementation

```
implicit def RectangleGeometry[T:Show] = new Geometry[T] {
  type S = Rectangle[T]
  override def perimeter(self: Exp[S]) =
    Fn()
      .name("perimeter")
      .self("self" ->> self)
      .returns[T]
      .body { args =>
        args("self").f("w") + args("self").f("w") +
        args("self").f("h") + args("self").f("h")
      }
}
def RectangleGeometryDeclaration =
  Impl()
    .bounds("<A:Add<Output=A>>+Copy")
    .inst(RectangleGeometry[A])
    .self[Rectangle[A]]
    .member(_.perimeter(PH).decl)
```

Listing 57 declares a `test` function which implicitly requests evidence for the implementation of the `Geometry` trait for a generic type. The purpose is to test whether trait resolution works.

Listing 57 Demo - Trait resolution

```
def test[T:Show](x: Exp[T])(implicit ev: Geometry[_]{type S = T}) =
  Fn()
    .name("test")
    .bounds("<A: Geometry<B>, B>")
    .arg("x" ->> x)
    .body( args =>
      ev.perimeter(args("x")) :#
    )
```

Listing 58 implements the `main` function which declares a local variable `i` used for creating a `Rectangle`, which is then passed to `test`.

Listing 58 Demo - Main function

```
def main =
  Fn()
    .name("main")
    .body { _ =>
      Let(Lit[i32]("3")) { i =>
        Let(Rectangle(i, i)) { r =>
          test(r) :#
        }
      }
    }
```

Finally, listing 59 inserts the items into a `File` and generates code to `demo.rs`.

Listing 59 Demo - Driver Scala method

```
def driver() = {
  val pw = new PrintWriter(new File("demo"))

  File(
    Use("std::ops::Add;"),
    Rectangle[A](PH,PH).decl,
    Geometry.decl,
    RectangleGeometry.decl,
    test[A](PH).decl
    main.decl,
  ).show(pw)

  pw.close()
}
```

The generated code, after formatting, is displayed in Listing 60. Neither the DSL nor rustc reports any errors when compiling the code.

Listing 60 Demo - Generated code

```
use std::ops::Add;
struct Rectangle<A> { w: A, h: A }
trait Geometry<A> {
  fn perimeter(self) -> A;
}
impl<A:Add<Output=A>+Copy> Geometry<A> for Rectangle<A> {
  fn perimeter(self) -> A {
    (((self.w + self.w) + self.h) + self.h)
  }
}
fn test<A: Geometry<B>, B>(x: A) -> () {
  x.perimeter(); ()
}
fn main() -> () {
  {
    let x1 = 3;
    {
      let x2 = Rectangle { w: x1, h: x1 };
      inner(x2); ()
    }
  }
}
```

8 Discussion

The discussion is divided into three parts. The first part examines the technical limitations of the implementation and possible improvements. The next part assesses both DSLs in relation to the design goals. The final part discusses how the product of the thesis could be useful to CDA.

8.1 Technical details

The evaluation for the shallow DSL shows that only the runtime interpolator, with calls to *rustc* is able to pass all tests. The runtime interpolator is however also the most limited as it only accepts whole files as input, at runtime. When instead using *syn*, the interpolators fail to detect type errors. Moreover, the *syn* parser gives little advantage over using IntelliJ's language injection, since language injection will, in addition to parsing, also do linting and name binding. A different approach would be to generate a Rust parser which parses a Rust AST, and analyze it semantically through macros. At present time, the official grammar at the Rust repository is however not yet canonical and may therefore not work as expected.

The compile time interpolators return an AST string literal, but it appears the AST is not propagated to other macro calls. The solution which libraries such as Quill employ, for communicating between macros, is to encode AST information inside of types [76]. The idea is to move information from the AST into the symbol table, which is accessible by all macros. In Quill, everything is a whitebox macro which produces an AST with an associated type annotation. The type annotation encodes information about the AST. Other macros can then access the AST by decoding the type annotation.

The deeply embedded DSL code is more verbose, and is currently limited in the number of constructs it can express, but should have more potential than the shallow DSL. One possibility is to turn the AST into a UAST by adding interpretations to other languages, such as C or C++. The DSL might even be able to introduce new semantics to the languages. For instance, C's lack of generics could be compensated for by generating monomorphic code. A generic struct in Rust could be expressed in C by generating duplicates of it. Unifying Rust with C++ is more difficult since their concepts do not map well to each other.

Coercions are missing in the current version of the deeply embedded DSL. Currently, it is not possible to pass a value in place of a reference, to a function expecting a reference. In Rust, the value would be referenced automatically. This behavior could be imitated in Scala by declaring an implicit conversion from values to references.

Another missing feature are patterns, which are a pre-requisite for enums and match expressions. While implicits could be used to verify if a pattern matches an expression, a difficulty is extracting the matching variables from the pattern. An idea is to express all patterns using Rust's '`@`' operator. The '`@`' operator binds the match of a pattern to a single variable. For example, given a `struct Foo(i32,i32)`, the pattern `Foo(x,y) => x+y` could be expressed as `foo @ Foo(_,_) => foo.x+foo.y`. This is easier to work with since only one variable needs to be taken into consideration.

Structural types, i.e., tuples, arrays, and slices, are also missing. Whereas tuples could be written as plain `HLists`, arrays are more difficult since their size is part of the type. For example, the type of an array with five integers is declared in Rust as `[i32,5]`. Creating a unique type for each size, such as '`_5`', is one solution. Another is to express the size as a singleton type literal. When Scala's establishes first-class language support for these, it should be possible to write the size directly as '`5`'.

Nothing currently prevents an immutable place expression from being mutated. Mutability is not tracked, but could be by embedding additional type information into expressions. A mutable expression could be expressed as `Mutable extends Exp[T]`, and an immutable as `Exp[T]`. Thereby, any mutable expression can be converted to immutable, but not the other way around.

8.2 Design goals

The shallow embedded DSL covers all of Rust's syntax and semantics. A downside is how static checking is restricted to string literals which are whole compilation units. However, the benefit of string literals is the consistency between the written and generated code, since everything is generated exactly as written. Combined with linting from language injection, it should also be easy for the user to detect syntax errors and name clashes.

The deeply embedded DSL is held back by the host language, and as a result cannot cover all of Rust's semantics. Macros may possibly be used to change Scala's semantics into something which accommodates for Rust's. While static checks are only enforced up until type checking, this might be enough for most tasks. In terms of ease-of-use, the combination of ADTs and fluent interfaces makes the code relatively easy to read and write. However, it is not always obvious what will be generated.

8.3 CDA

CDA will use the Rust DSL to translate incoming queries into low level Rust code at runtime. Queries will include information about both the logic and types of the program to be generated. For example, a basic query may request the summation of a stream of integers. As this information is not available at compile time, the DSL must allow for some form of runtime metaprogramming. The shallow embedded DSL can freely alter both the logic and types of the generated program by splicing strings with arguments. Type checking for the spliced strings is dynamic as it can either be performed at runtime or compile time. On the other hand, the deeply embedded DSL is statically typed, meaning all types must be known at compile-time. This is an impossible requirement to satisfy. Although less flexible than splicing, it is still possible to override what code a type generates to with implicits.

An issue with regards to generating and executing programs at runtime, which was not addressed, is error reporting. Whenever an error occurs it must be reported back to the user.

This involves back-tracking, from the place the error occurred in the generated code, to where the logic was first specified in the front-end code. The benefit of generating to Rust is how most errors are caught early on during compilation.

Code written in the DSL will have to make calls to various libraries and external crates. The shallow embedded DSL might therefore require integration with cargo to fetch and statically check against crates. As for the deeply embedded DSL, an option is to declare or generate facades to exterior interfaces.

Most problems regarding both DSLs could be solved by combining the two into one. Integration with other DSLs, such as Weld, could also be explored. When one DSL is insufficient to solve a problem, the other could step in and help. An advantage of the deeply embedded DSL is its ability to type check Rust AST nodes within Scala at compile time. Meanwhile, the shallow embedded DSL has potential for runtime metaprogramming. The DSLs could be integrated by extending the string interpolator of the shallow DSL with the ability to splice AST nodes of the deeply embedded DSL. Conversely, the interpolated result could be inserted as a node into the AST. The reason for why the current version does not support this feature is due to how code is being generated. The shallow embedded DSL recursively concatenates strings, while the deeply embedded DSL prints code directly to a file. The latter approach is less expensive but also inflexible as the generated code for each individual node is not kept in memory. This makes it impossible to run external static checks on a per-node basis.

The closing question is whether the ideas behind the DSLs will work in practice for CDA, or if there is a better approach to runtime code generation. Generating to Rust is beneficial as it enforces static safety. On the other hand, coding Rust directly in Scala might be complicated, as Rust is low-level. Therefore, a possible idea is to write a DSL which provides a small set of building blocks that abstract over Rust's syntax and semantics. This would be similar to how MonetDB's Voodoo DSL renders an abstraction layer above OpenGL [22]. By abstracting over Rust, it would be possible to generate programs, specific to CDA's needs, without having worrying about satisfying Rust's strict static checks. The tradeoff is flexibility, as abstractions lose control over the details.

9 Conclusion

Code generation DSLs will be a crucial component in the next generation of distributed systems, for exploiting the resources of heterogeneous hardware. Rust's safety guarantees, combined with its zero cost abstractions, makes it a promising target language. Meanwhile, Scala's rich type system and flexibility for embedding DSLs makes it a favorable host language.

This thesis has implemented and evaluated two DSLs for generating Rust code in Scala, as part of the CDA distributed system. The aim is for CDA to use the DSLs for translating incoming queries into low-level Rust code at runtime. The first DSL is a string interpolator, with ability to splice and statically verify snippets of Rust code, using external processes, at either compile time or runtime. The second DSL builds an AST using ADTs, and is able to express custom functions, structs, and traits using fluent interfaces and HLists.

Evaluation shows the first DSL is limited in terms of static checking of Rust snippets which represent sub-parts of Rust programs. Secondly, its ability to merge snippets of code at compile time is restricted to only being able to splice string literals. The second DSL only supports a subset of Rust features. Mainly, it lacks syntax for expressing enums, structural types and patterns, and semantics for coercions, mutability checking and borrow checking.

The former DSL has the potential for metaprogramming at runtime. This is necessary since the logic and type information of incoming queries is unknown at compile time. The second DSL instead focuses on building an IR, which could conceivably be used to generate code into other languages than Rust. As future work, the main interest is to merge the DSLs into one, in order to combine their advantages. Integrating and testing the DSLs with the CDA system is also on the agenda. Secondary objectives are to improve the first DSL with the ability to splice non-literal arguments at compile-time, and to extend the second DSL with more syntax, and possibly semantics through the use of macros.

Conclusively, the true impact and usefulness of the DSLs is yet to be decided. Optimally, their application will result in a system capable of meeting the rigorous performance demands of continuous deep analytics, and allow for new technologies to emerge.

References

[1] M. R. technopedia, “Deep analytics definition,” 2011. [Online]. Available: http://searchbusinessanalytics.techtarget.com/definition/deep-analytics. [Accessed: 23-Mar-2018]

[2] RISE SICS, “Continuous deep analytics is granted 33 million sek by ssf,” 2017. [Online]. Available: https://www.sics.se/media/news/continuous-deep-analytics-is-granted-33-million-sek-by-ssf. [Accessed: 23-Mar-2018]

[3] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, *Deep learning*, vol. 1. MIT press Cambridge, 2016.

[4] S. Palkar, J. Thomas, D. Narayanan, A. Shanbhag, R. Palamuttam, H. Pirk, M. Schwarzkopf, S. Amarasinghe, S. Madden, and M. Zaharia, “Weld: Rethinking the interface between data-intensive applications,” *arXiv preprint arXiv:1709.06416*, 2017.

[5] H. Chafi, Z. DeVito, A. Moors, T. Rompf, A. K. Sujeeth, P. Hanrahan, M. Odersky, and K. Olukotun, “Language virtualization for heterogeneous parallel computing,” in *ACM sigplan notices*, 2010, vol. 45, pp. 835–847.

[6] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, “Spark: Cluster computing with working sets.” *HotCloud*, vol. 10, nos. 10-10, p. 95, 2010.

[7] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, “Apache flink: Stream and batch processing in a single engine,” *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.

[8] M. Grossman, S. Imam, and V. Sarkar, “Hj-opencl: Reducing the gap between the jvm and accelerators,” in *Proceedings of the principles and practices of programming on the java platform*, 2015, pp. 2–15.

[9] F. McSherry, M. Isard, and D. G. Murray, “Scalability! But at what cost?” in *HotOS*, 2015.

[10] G. M. Essertel, R. Y. Tahboub, J. M. Decker, K. J. Brown, K. Olukotun, and T. Rompf, “Flare: Native compilation for heterogeneous workloads in apache spark,” *arXiv preprint arXiv:1703.08219*, 2017.

[11] J. Svenningsson and E. Axelsson, “Combining deep and shallow embedding for edsl,” in *International symposium on trends in functional programming*, 2012, pp. 21–36.

[12] J. Gibbons and N. Wu, “Folding domain-specific languages: Deep and shallow embeddings (functional pearl),” in *ACM sigplan notices*, 2014, vol. 49, pp. 339–347.

[13] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer, “RustBelt: Securing the foundations of the rust programming language,” *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, p. 66, 2017.

[14] D. Wampler and A. Payne, *Programming scala: Scalability = functional programming+ objects*. " O’Reilly Media, Inc.", 2014.

[15] The Apache Software Foundation, “Powered by Flink,” 2017. [Online]. Available: https://flink.apache.org/poweredby.html. [Accessed: 02-Jun-2018]

[16] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, “Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing,” in *Proceedings of the 9th usenix conference on networked systems design and implementation*, 2012, pp. 2–2.

[17] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, and others, “Apache spark: A unified engine for big data processing,” *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, 2016.

[18] A. Grove, “DataFusion,” 2018. [Online]. Available: https://www.datafusion.rs. [Accessed: 09-May-2018]

[19] T. G. Stanislav Bohm Vojtech Cima, “Rain,” 2018. [Online]. Available: https://github.com/substantic/rain. [Accessed: 09-May-2018]

[20] A. Mahgoub, S. Ganesh, F. Meyer, A. Grama, and S. Chaterji, “Suitability of nosql systems—cassandra and scylladb—for iot workloads,” in *Communication systems and networks (comsnets), 2017 9th international conference on*, 2017, pp. 476–479.

[21] Apache Software Foundation, “Apache arrow,” 2018. [Online]. Available: https://arrow.apache.org. [Accessed: 09-May-2018]

[22] H. Pirk, O. Moll, M. Zaharia, and S. Madden, “Voodoo-a vector algebra for portable database performance on modern hardware,” *Proceedings of the VLDB Endowment*, vol. 9, no. 14, pp. 1707–1718, 2016.

[23] B. Nystrom, “Crafting interpreters,” 2018. [Online]. Available: http://craftinginterpreters.com. [Accessed: 03-May-2018]

[24] G. Hjort Blindell, “Universal instruction selection,” PhD thesis, KTH Royal Institute of Technology, 2018.

[25] S. P. JONES, “2 lexical structure,” *Journal of Functional Programming*, vol. 13, no. 1, p. 7, 2003.

[26] W. A. Andrew and P. Jens, *Modern compiler implementation in java*. Cambridge, 2002.

[27] G. K. Pullum and G. Gazdar, “Natural languages and context-free languages,” *Linguistics and Philosophy*, vol. 4, no. 4, pp. 471–504, 1982.

[28] A. Ranta, *Implementing programming languages. An introduction to compilers and interpreters*. College Publications, 2012.

[29] A. Belevantsev and A. Avetisyan, “Multi-level static analysis for finding error patterns and defects in source code,” in *International andrei ershov memorial conference on perspectives of system informatics*, 2017, pp. 28–42.

[30] *Compilers : Principles, techniques, & tools*, 2. ed.. Boston: Pearson Addison-Wesley, 2007.

[31] M. H. Sørensen and P. Urzyczyn, *Lectures on the curry-howard isomorphism*, vol. 149. Elsevier, 2006.

[32] P. Jalote, *An integrated approach to software engineering*. Springer Science & Business Media, 2012.

[33] University of Washington, "CSE341: Programming Languages," 2013. [Online]. Available: https://courses.cs.washington.edu/courses/cse341/13wi/unit6notes.pdf. [Accessed: 02-Jun-2018]

[34] F. Pfenning and C. Elliott, "Higher-order abstract syntax," in *ACM sigplan notices*, 1988, vol. 23, pp. 199–208.

[35] A. Balasubramanian, M. S. Baranowski, A. Burtsev, A. Panda, Z. Rakamari, and L. Ryzhyk, "System programming in rust: Beyond safety," *ACM SIGOPS Operating Systems Review*, vol. 51, no. 1, pp. 94–99, 2017.

[36] The Rust Team, "The rust programming language," 2017. [Online]. Available: https://doc.rust-lang.org/reference. [Accessed: 27-Mar-2018]

[37] *Programming rust: Fast, safe systems development*, 1. ed.. O'Reilly Media, 2017.

[38] The Rust Team, "The rust programming language, first edition," 2017. [Online]. Available: https://doc.rust-lang.org/book/first-edition. [Accessed: 27-Mar-2018]

[39] The Rust Team, "Commit 41a21f053ced3df8fe9acc66cb30fb6005339b3e - remove typestate from code, tests, and docs," 2012. [Online]. Available: https://github.com/rust-lang/rust/commit/41a21f053ced3df8fe9acc66cb30fb6005339b3e. [Accessed: 09-May-2018]

[40] The Rust Team, "Finish runtime removal," 2014. [Online]. Available: https://github.com/rust-lang/rust/pull/18967. [Accessed: 09-May-2018]

[41] T. B. L. Jespersen, P. Munksgaard, and K. F. Larsen, "Session types for rust," in *Proceedings of the 11th acm sigplan workshop on generic programming*, 2015, pp. 13–22.

[42] The Rust Team, "Non-lexical lifetimes," 2017. [Online]. Available: https://github.com/nikomatsakis/nll-rfc/blob/master/0000-nonlexical-lifetimes.md. [Accessed: 26-Mar-2018]

[43] The Rust Team, "The rust programming language, second edition," 2017. [Online]. Available: https://doc.rust-lang.org/book/second-edition. [Accessed: 27-Mar-2018]

[44] The Rust Team, "Rust subtyping," 2014. [Online]. Available: https://doc.rust-lang.org/beta/nomicon/subtyping.html. [Accessed: 27-May-2018]

[45] Alexis Beingessner, "Learning rust with entirely too many linked lists," 2015. [Online]. Available: http://cglab.ca/~abeinges/blah/too-many-lists/book/. [Accessed: 27-Mar-2018]

[46] The Rust Team, "Rustc guide," 2018. [Online]. Available: https://rust-lang-nursery.github.io/rustc-guide/. [Accessed: 18-Apr-2018]

[47] The Rust Team, "Rust subtyping," 2017. [Online]. Available: https://github.com/rust-lang/rust/blob/master/src/grammar/raw-string-literal-ambiguity.md. [Accessed: 28-May-2018]

[48] The Rust Team, “Rust - canonical grammar,” 2015. [Online]. Available: https://github.com/nagisa/rfcs/blob/grammar/text/0000-grammar-is-canonical.md. [Accessed: 17-Apr-2018]

[49] The Free Software Foundation, GNU Operating System, “Bison,” 2014. [Online]. Available: https://www.gnu.org/software/bison. [Accessed: 17-Apr-2018]

[50] V. Paxson, “Flex,” 2018. [Online]. Available: https://github.com/westes/flex. [Accessed: 17-Apr-2018]

[51] JFlex Team, “JFlex,” 2018. [Online]. Available: https://github.com/jflex-de/jflex. [Accessed: 17-Apr-2018]

[52] P. T. Ivan Kochurkin, “Unification in chalk,” 2016. [Online]. Available: http://smallcultfollowing.com/babysteps/blog/2017/03/25/unification-in-chalk-part-1/. [Accessed: 23-Apr-2018]

[53] P. VAN WEERT, “EXTENSION and optimising compilation of constraint handling rules,” 2010.

[54] M. Carro, “Constraint logic programming,” 1987.

[55] I. Lightbend, “Tour of scala,” 2017. [Online]. Available: https://docs.scala-lang.org/tour/tour-of-scala.html. [Accessed: 06-Jun-2018]

[56] A. Ivanov, “Default type parameters with implicits in scala,” 2017. [Online]. Available: https://www.cakesolutions.net/teamblogs/default-type-parameters-with-implicits-in-scala. [Accessed: 09-Jun-2018]

[57] C. Miles Sabin, “Feature overview: Shapeless 2.0.0,” 2016. [Online]. Available: https://github.com/milessabin/shapeless/wiki/Feature-overview:-shapeless-2.0.0. [Accessed: 09-Jun-2018]

[58] C. Miles Sabin, “Shapeless hlist operations,” 2018. [Online]. Available: https://github.com/milessabin/shapeless/blob/master/core/src/main/scala/shapeless/ops/hlists.scala. [Accessed: 28-May-2018]

[59] D. Gurnell, *The type astronaut’s guide to shapeless*. Lulu. com, 2017.

[60] C. Miles Sabin, “Shapeless labelled,” 2018. [Online]. Available: https://github.com/milessabin/shapeless/blob/master/core/src/main/scala/shapeless/labelled.scala. [Accessed: 09-Jun-2018]

[61] J. Z. George Leontiev Eugene Burmako, “Literal-based singleton types,” 2017. [Online]. Available: https://docs.scala-lang.org/sips/42.type.html. [Accessed: 27-May-2018]

[62] C. Frank S. Thomas, “Refined - simple refinement types for scala,” 2018. [Online]. Available: https://github.com/fthomas/refined. [Accessed: 27-May-2018]

[63] P. Riti, *Practical scala dsls: Real-world applications using domain specific languages*. Berkeley, CA: Apress, 2018.

[64] A. Munshi, “The opencl specification,” in *Hot chips 21 symposium (hcs), 2009 ieee*, 2009, pp. 1–314.

[65] JetBrains, “Using language injections,” 2018. [Online]. Available: https://www.jetbrains.com/help/idea/using-language-injections.html. [Accessed: 30-May-2018]

[66] C. Marek Kaput JetBrains, “IntelliJ rust,” 2016. [Online]. Available: https://www.jetbrains.com/help/idea/using-language-injections.html. [Accessed: 30-May-2018]

[67] D. Shabalin, E. Burmako, and M. Odersky, “Quasiquotes for scala,” 2013.

[68] L. Parreaux, A. Shaikhha, and C. E. Koch, “Squid: Type-safe, hygienic, and reusable quasiquotes,” in *Proceedings of the 8th acm sigplan international symposium on scala*, 2017, pp. 56–66.

[69] Guillaume Martes, “Generalized Algebraic Data Types in Scala,” 2018. [Online]. Available: https://gist.github.com/smarter/2e1c564c83bae58c65b4f3f041bfb15f. [Accessed: 02-Jun-2018]

[70] J. Carette, O. Kiselyov, and C.-c. Shan, “Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages,” *Journal of Functional Programming*, vol. 19, no. 5, pp. 509–543, 2009.

[71] A. Kennedy and C. V. Russo, “Generalized algebraic data types and object-oriented programming,” *ACM SIGPLAN Notices*, vol. 40, no. 10, pp. 21–40, 2005.

[72] P. Wadler, “The expression problem,” *Java-genericity mailing list*, 1998.

[73] V. C. Martin Odersky Philippe Altherr, “The Scala language specification - Version 2.12.” 2016.

[74] The Rust Team, “Operators and symbols,” 2018. [Online]. Available: https://doc.rust-lang.org/book/second-edition/appendix-02-operators.html. [Accessed: 02-Jun-2018]

[75] David Tolnay, “Syn - TraitItem,” 2018. [Online]. Available: https://dtolnay.github.io/syn/syn/enum.TraitItem.html. [Accessed: 02-Jun-2018]

[76] F. Brasil, “Quill,” 2018. [Online]. Available: http://getquill.io. [Accessed: 11-Jun-2018]

TRITA TRITA-EECS-EX-2018:171