Dynamic Rust Code-Generator

OSCAR BJUHR

Master’s Programme, Software Engineering of Distributed Systems
Date: March 16, 2018
Supervisor: Paris Carbone, Lars Kroll
Examiner: Christian Schulte
Swedish title: Dynamisk Rust Kod-Generator
School of Electrical Engineering and Computer Science

Abstract

Sammanfattning

Contents

Listings

List of Figures

Chapter 1

Introduction

Modern software development in the Big Data domain focus on scale-out performance. Instead of improving the capacity of a single computer, called scale-up, the application will run on a large cluster of possibly distributed machines. Coordination in such a parallel distributed setting is non-trivial and error prone. Therefore, most developers turn to abstract high-level framework for parallel programming where the distribution is implemented implicitly by the framework [2] [8].

Using such a framework will make it easier for developers with limited experience of distributed parallel programming to build programs which execute on a cluster. The benefit of using a cluster as execution platform is performance scalability [13]. Most state-of-art distributed frameworks run on the Java Virtual Machine (JVM) [2] [8]. This will work fine on a homogeneous cluster but it will be inefficient on a heterogeneous cluster. A heterogeneous cluster contains different types of hardware accelerators, such as graphics processing unit (GPU) and field-programmable gate array (FPGA), combined with central processing units (CPUs).

To fully utilise the potential of such a cluster, the code would need to be specifically tailored for the underlying hardware of each individual node. This is a continuous problem since vendors of accelerators tend to update and release new application programming interfaces (APIs) to enable more efficient use of their hardware. If an application explicitly use the API it would not be portable [22]. Making sure that an application continues to be portable to new hardware and APIs would require substantial continuous manual efforts by the application developer. Using a more high-level framework for distributed parallel computing, the responsibility of continuous portability maintenance is instead shifted to the developer of the framework. Thus, the application developer will be alleviated of this strenuous work effort [11].

Deep Neural Networks (DNNs) is a machine learning model depending on effective computations of floating point operations. GPUs are commonly used for executing DNNs due to their superior parallel computing power. An architecture which has an emerging potential for substituting and complementing GPUs are FPGAs [29]. The main advantage of FPGAs is their energy efficiency [3]. This indicates that the components in a state-of-art heterogeneous cluster is subject to change in the future.

1.1 Background

The rise of distributed heterogeneous execution platforms brings the need for abstract high-level programming framework [36]. The distribution of resources in a cluster is usually handled by a resource negotiator such as YARN [40] and Mesos [19]. This decouples the resource management from the application logic.

To avoid run-time performance penalties and to reduce the risk of run-time exceptions the application can be used as a meta-program, i.e. program generator. The input to the program generator usually resembles an abstract syntax tree (AST). Assuming that the meta-program and AST is well-typed and semantically correct, a program generator can transform it into low-level instructions and compiled to native code [36]. When targeting a heterogeneous platform, the node responsible for building and compiling needs to be able to cross compile. A cross compiler is a compiler which is it-self compiled to execute on the host platform but emits code for another target platform [39]. Platform is a joint description for the underling machine and its software, such as operating system, hardware architecture, and instruction set.

The compiling process can be divided into two course grained parts, source code to IR and IR to native code. The IR avoids assumptions regarding the target platform as much as possible. It should also be uncoupled to the actual source code, being an abstract representation of the source codes meaning, i.e. semantics. The IR is used to emit code to a specific underlying platform, generating machine instructions for the specific target platform [1]. This step is composed of two parts, generating object files and linking objects file to create an executable [34].

Light-weight modular staging (LMS) [32] with language virtualization [11] has been used in Delite [22] to generate code for a heterogeneous platform. The underlying conceptual idea of LMS is similar to RDDs in Spark [42] where operations are lazily evaluated. In Delite, an intermediate representation (IR) for the program is constructed during run-time. This IR is a kind of AST and is called the staged representation of the program. It is equivalent to the execution plan of Spark RDDs. The final staged representation of the program is used to emit native code for each target platform in the heterogeneous cluster [22].

1.1.1 Continuous Deep Analytics

The project Continuous Deep Analytics(CDA) aims to tackle the need for computationally heavy analysis of data-streams, such as DNNs in machine learning. Therefore, CDA needs to be able to fully utilise the computational capabilities of a heterogeneous cluster. However, this should not be done at the expense of requiring manual low-level implementation of the user. CDA will follow Spark and Flink's concept of supplying high-level abstraction to the user and implicitly handling the parallelization and distribution of the work effort.

CDA is in the starting phase so few concrete definitions of the system have been made. The initial prototype is similar to Flink [8], it is composed of a Client, Driver, and Worker. The Driver corresponds to Flink's Job Manager, and is responsible for splitting and distributing the work effort as well as monitoring the cluster. The tasks will have to be compiled to native code for the workers to fully utilise the underlying heterogeneous cluster. This will also be part of the Drivers responsibility. The workers will be responsible for executing the tasks.

The Client is a front-end for the user. Implementation for the front end will be available in several programming languages. The user will specify the stream processing application in the front-end. The application will then be compiled to an intermediate representation (IR) which will then be sent to the Driver. The IR is going to be optimised e.g. using data-flow optimisations. This can either happen at the client or at the Driver.

The Driver is the responsible for splitting the application into tasks,

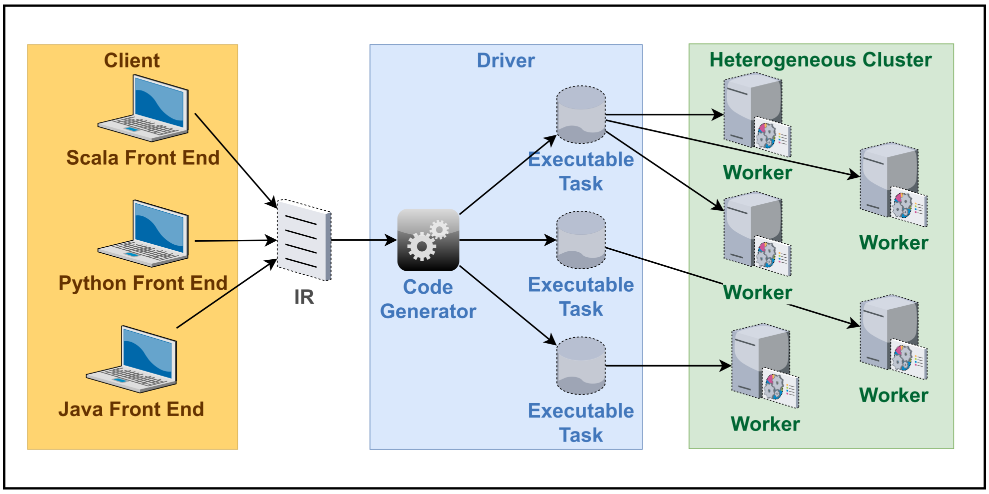

Figure 1.1: Overview of the Continuous Deep Analytics System

mapping the tasks to specific workers in the cluster, and interpreting the IR of each task to hardware specific native code for the workers which are going to execute the task.

1.2 Problem

CDA aims to move from Java and the JVM to increase performance. The problem with shifting to a high-performance programming languages which is compiled to native code is the loss of many safety guarantees. This has been known as the holy-grail of programming language domain, a language can either supply low-level control to increase performance, or statically guarantee run-time safety of a compiled program. Rust is a new programming language which claims to achieve both sides of this problem [24]. Therefore, CDA want to be able to generate Rust programs from the IR. The problem with Rust is that it is still not in a stable state. Rust retains the ability to do breaking updates by declaring it to be in a unstable state. Breaking updates are updates which may not guarantee backwards compatibility of new versions. Programs written in an older version of Rust may be invalidated by a later release of Rust. Also, the IR in CDA is not defined and is guaranteed to vary and expand from the prototype IR used during this thesis.

The research question is therefore: how can an interpreter of the IR to Rust be implemented to easily adapt to future changes of both Rust and the IR?

1.3 Purpose

The purpose of the thesis is to show-case how a code-generator should be composed to generate Rust code. The main reason is Rust's high performance and good static safety-guarantees. This will make a stream processor high-performing whilst reducing the risk of run-time exceptions.

1.4 Goal

The goal of the thesis is a prototype Rust code generator which emits code for a set of target platforms.

1.4.1 Benefits, Ethics and Sustainability

A high-level frame-work which enables the user of to easily migrate their application to a more energy efficient cluster, e.g. substitute GPUs with FPGAs [29], can have a positive effect on energy consumption. Data-centers for co-located cloud systems [23], i.e. a company rents a part of a data-center's capacity, may benefit. The company hosting the data-center can lower energy consumption by switching to more energy efficient hardware components. This in turn may let them lower their rent to gain leverage over competition without incurring any financial loss.

As massive internet of things (IoT) becomes part of reality, frameworks for handling huge amount of high velocity data-streams become essential. Analysing and reacting to the data in a correct way, and adapting to the performance need on demand will be an important aspects of massive IoT [41].

Being able to think in a high-level of abstraction will enable the developers of such systems to argue and prove the correctness of their application. Higher-level of abstraction also makes it easier for different domain experts with limited programming experience of distributed systems to understand the implementation of such applications [20].

Applications for smart cities should also be robust, since applications supporting the core of society must have high availability [41]. Rust with it's high performance and safety-guarantees [5] is a good programming language for such applications.

1.5 Related Work

Spark is an expressive framework for distributed parallel computing running on the JVM. Spark aims to improved performance by avoiding moving data eagerly to external stable storage. Other popular distributed computing frameworks prior to Spark moved all intermediate result to an external stable storage, such as a distributed database, to guarantee fault tolerance. This impedes their execution and decrease performance significantly. Spark introduced Resilient Distributed Datasets (RDDs) as an alternative. RDDs represent an abstract execution plans of operation on data stored in stable storage. RDDs are lazily evaluated. No intermediate results are calculated before a result from the expression the RDD represents is explicitly requested by the user. At the time when the result value is requested, Spark will optimise the execution plan and distribute the workload over the cluster. Spark improved performance of distributed parallel computing significantly, up to 20x compared to Hadoop [42].

1.5.1 Spark and Native Code Generation

Application analysing Big Data tend to utilise a mixture of procedural algorithms and relational queries. Using Sparks RDDs or other distributed computing frameworks, as Hadoop, for relational queries can be tedious and complex. It requires a lot of manual optimisation to match the performance of frameworks specialise for relational queries. To tackle this problem, Spark introduced the DataFrame concept along with an optimiser specifically developed for query optimisation called Catalyst. DataFrames represents a distributed set of rows with a uniform schema such as a table in a relational database. These rows can be manipulated using the DataFrame API, which consists of a set of lazily evaluated relational operators. These relational operators correspond to queries in the Structured Query Language (SQL). A DataFrame can be transformed to an RDD of row objects. This bridge the gap between relational queries and procedural algorithms [2].

Sparks performance can suffer radically due to the intent to be expressive whilst keeping the distribution “under the hood”. Flare is an attempt to fix a part of the performance issue, focusing on DataFrames. Spark operations on a DataFrame will originally be executed on the JVM. Running operations on a virtual machine (VM) is inherently slower than native-code. Therefore, Flare transforms relational operators on DataFrames to native code. This improved performance significantly. Sparks query performance with Flare matched the performance of the best SQL engines without affecting Sparks expansive expressiveness [14]. Flare is a showcase of the potential of moving from interpreting instructions by a VM to compiling to native code.

1.5.2 Flink, a Stream Processing Framework

Flink [8] is an open source system focused on continuous distributed computing. A Flink program will be compiled and scheduled once and then run for a long period of time. Therefore, Flink needs to be adaptive and fault tolerant during run time. To achieve this, Flink employs a dynamic cluster architecture which is composed of three processes: the client, the Job Manager and the Task Managers [9]. The client will compile and optimise the logical pipeline before sending it to the Job Manager. The Job Manager is responsible for coordination, physical deployment and fault tolerance. The Task Managers are workers, using a single JVM to execute tasks assigned to it. The system can even handle Job Manger failures using leader election in ZooKeeper [21].

1.5.3 YARN and Stream Processing

YARN [40] and Mesos [19] can enable multiple frameworks to co-exist in the same cluster. Resources gets distributed by the resource negotiator, i.e. YARN or Mesos. The framework which uses the resource is unaware of what other frameworks co-exists in the cluster and is oblivious of the resource management. This makes developing frameworks for distributed parallel computing possible without having to implementing an specific resource manager. It also handles fair resources distribution between different frameworks without the need for an explicitly interaction between the frameworks.

However, the YARN scheduler is developed for batch-processing where a job will request a static set of containers and use them for a limited time before releasing them. It is not developed for the dynamic nature of stream processors, where the input velocity and performance needs can fluctuate greatly. But with some effort it is possible to seamlessly run a stream processor on top of YARN [28]. This enables the stream processor evict input data to storage if the processing needs becomes to great. The evicted data can later be processed in batches when the input streams velocity has decreased.

1.5.4 Meta-Programming and Code Generation

Both Spark and Flink are examples of frameworks which enable the developer to focus on higher-level logical meanings of a program, high-level semantics, without requiring detailed knowledge of the underlying low-level concrete implementations. This is a form of Meta-Programming, where low-level implementations will be generated based on a higher-level abstract specification supplied by the user [36].

The problem solved by meta-programming shares some commonality with General-Purpose Languages (GPLs). A GPL supply the developer with a abstract human-readable language which the compiler in turn converts to a representation which can be interpreted by the underlying hardware. Examples of GPLs are C, C++ and Java.

Meta-programming languages is a broad category and spans over a large variety of language techniques. It can be incorporated into a GPL, using macros, templates, or data-structures to represent the intended program. Alternatively, a narrow language can be developed for the specific application domain. This is call a Domain Specific Language (DSL) [36].

Domain Specific Languages for High Performance Stream Processing

Ziria is an example of a DSL. The implementation of wireless protocols is complex due to the high throughput demands. Data in the rate of Gigabits/second have to be processed by a wireless router. Most protocols have therefore been implemented without much hardware abstractions. This enhances performance, since the programs are tailor-made for the underlying hardware, but it makes the software development process more complex and error prone. Ziria enhances the development and maintenance of wireless protocols by exposing high-level abstraction to the user. The abstraction consists of two main components: stream transformer and stream computer. A stream pipeline is built using these two components. Ziria aims to not restrict the domain experts expressiveness. To enhance performance, Ziria apply aggressive domain specific optimisation to the program and then compile it to highly optimised hardware specific native code. Ziria is an external DSL which means that the developers implemented a compiler, parser and similar tools specifically for Ziria. Implementing these tools requires a substantial effort. Instead a DSL can be embedded into a host language to reduce the work effort by reusing the tools implemented for the host languages [20].

Lime Java [4] is another implemented DSL for stream processing on a heterogeneous cluster. Lime Java is integrated into Java and has a set of language constructs which limits the usage of global fields and static fields. It also encourages the usage of none mutable types. The constructs which enable these restrictions on methods are two keywords, local and global. A local method can only invoke other local methods while global methods may invoke either local or global methods. The local methods have the restriction that they may only access the instance's local fields, but can be part of a stateful instance. The local method can use the instance's local fields though Java's "this" construct, with the restriction that it is not a static field. The two constructs enables Lime to represent a stream pipeline into a set of autonomous tasks, where each task is a set of local methods. Hence, these tasks are guaranteed to not be in need of any synchronisation. The tasks can therefore be dynamically allocated to hardware components in the cluster. No distributed coordination aside from input-output pipelining need to be enforced between these tasks. To further decrease network traffic, connected tasks in the pipeline can be merged and assigned to a single node in the cluster. This also enables the compiler to distribute the workload efficiently and more evenly over the cluster. A running graph may be dynamically extended with new task. Added task will automatically be connected and started.

1.5.5 Delite

The goal of the Delite framework [22] is similar to CDA. It exposes high level abstractions to the user which still enable full utilisation of a heterogeneous platform. The program is converted to an IR which is subject to domain specific optimisations. The optimised IR is then used to emit code for the specific targets in the cluster. For CPUs, Delite emits Scala and C++ code which is compiled to either JVM byte-code or native code for the target platform. This makes Delite portable, it can always fallback to the JVM if the GCC cross compiler for a specific target platform does not work. For GPUs, Delite emits CUDA code which is compiled to native code for the specific target platform using CUDAs compiler. The CUDA compiler is natively a cross compiler, the compiler is not going to be executed on a GPU. Instead it relies on a C++ compiler for the host platform as a host compiler [30]. Thus, cross compiling CUDA code is supplied by nVidia.

1.6 Delimitations

The result of the thesis will be a prototype, a proof-of-concept, and will therefore be limited in its expanse. The front ends and the IR of CDA are not yet developed, their occurrence in the thesis is as prototypes as well. The development of the driver is outside the scope of this thesis, and is purely incorporated as a conceptual proof that the code-generator can cooperate with a generic driver.

1.7 Outline

Chapter 2

Theoretic Background

Meta-programming is a technique to model an intended program at a high-level of abstraction. An abstract program model may need the capability to modify it's own behaviour. This can either be done during compile-time or run-time. However, this is usually done during run-time and is called dynamic reflection [36].

2.1 Domain Specific Languages

GPLs are great tools for programmers who's work spans over several application domains. But a GPL might impede the development process for user which has limited experience with software development and who are specialised in a specific application domain. A more precise and narrow language may prove beneficial for such developers. Such languages are called DSLs. A good DSL should make the development process fast and efficient whilst still being correct and not limit the expressiveness within the application domain.

To develop such a DSL can become costly if it is done from scratch. To avoid this, embedded DSL takes advantage of the host language, the language the DSL interpreter is developed in. Much of the host language's syntax, semantics, development tools, and other related artefacts is reused by the DSL.

Focus during the development of the DSL can instead be shifted from syntax to the semantics of the domain specific parts of the DSL. This will reduce development cost of the DSL significantly and make it less error prone. The semantics of the DSL should be clear and capture the intuition behind the domain concepts to simplify the formal process of proving the correctness of a program [20].

2.1.1 Different embedded DSLs

Embedding of DSL is widely split into two categories, shallow and deep embedding. Shallow embedding is done by interpreting the DSL constructs as constructs directly in the target language. Even though interpreting instruction directly to target code incurs little performance overhead, the lack of ability for domain specific optimisation in a shallow DSL can mitigate performance gain by eagerly evaluating the DSL constructs. [25] showed that, by not eagerly evaluating an domain specific language, R in that case, performance can be significantly improved. By interpreting R code as an abstract syntax tree (AST), domain specific optimisation's may be applied more effectively. Thus, performance of their non-eager interpretation of a R program in Java could outperform the regular R interpreter which was implemented in C.

Shallow embedded DSLs require a partial-evaluator to apply domain specific optimisation's to the generated code. Implementation of a partial-evaluator requires a substantial effort.

Deep embedding circumvent this problem concerning domain specific optimisation by not eagerly evaluating the DSL constructs. Instead, the DSL constructs is transformed into an IR. The IR is implemented as a data-structure and domain specific optimisation's can be applied directly do it. The resulting, optimised IR is then interpreted to corresponding constructs in the target language [18].

2.2 Multi-Stage Programming Languages

Program generators is another effective way of enhancing domain specific application development. They enable high-level abstractions which can be used to argue the semantics of the program. It should, similarly to embedded DSLs, enable these abstractions without incurring any run-time overhead due to interpretation. In contrast with embedded DSLs, instead of interpreting constructs directly in the host language, multi-stage programming language [38] use the host language as a meta code-generator and exposes staged representations to the user. The host language code express how to create a representation of the program similar to an AST. This gives a more transparent construction of the staged program and the user will easily understand what actual representation is going to be created.

Embedded DSLs interpret host language constructs to a staged representation to avoid exposing the staged representation to the user. The main reason for this is that exposing the staged representation to the DSL user gives the user a feeling of writing a program generator instead of an actual program. The drawback of embedded DSL is that the user may find it hard to understand what the staged representation of the program will actually look like. Thus, it can be unclear what target code will be emitted for the program.

The staged AST representation enables validity checks of the semantics to verify it's correctness, as type-checking the AST. The AST can be optimised using domain specific knowledge. Source code in the target language is generated from the AST representation by the generator.

The execution of a program in a multi-staged programming language is composed of three stages; generation, compilation and execution. In comparison, execution of programs in embedded DSLs and GPLs are composed of two stages; compile-time and run-time.

The generation stage will interpret the staged programming language constructs, create the staged representation, and emit source code in the target language. The target source code is passed to the target language's compiler and compiled, e.g. to a binary executable. Lastly, the binary is executed. This is done dynamically, which means that the staged representation will be dynamically created based on the input to the program. This will enable the staged program to adapt more to the input data at the expense of dynamical staging and compilation of the actual binary. This will incur significant overhead costs during run-time if the format of the input data is inconsistent and changes a lot.

2.2.1 Language Virtualization Using the Stage Programming Model

Using a staged programming language can be complicated. Finding the desired classes representing staged operations can require reading through the library implementation. Requiring the user to explicitly access and construct the staged representation can easily become error prone. A technique for alleviating the user of this and making the library more user-friendly is Language Virtualization [11].

In a virtualized staged programming language, the operations available for a specific data-type is implemented as methods for the staged representation of the data-type. The methods will not be evaluated directly however, as could be expected by regular methods. Instead the methods return a lifted, i.e. staged, representation of the operation. This lifted representation is apt for domain specific optimisation. However, from the user's perspective this will not be clearly noticeable. The lifting into a staged representation and all optimisation are done implicit by the library. This is similar to what Spark's RDDs [42] does, creating a execution plan which is stalled until the result is explicitly requested by the user. Using a language which supports overloading of language operators, such as "+", the virtualization and lifting can be made to include host language language constructs. Thus, A + B can be lifted to Add(A, B). Scala is a language which support this kind of overloading [11].

2.2.2 Lightweight Modular Staging

The program generator can be reduced to a library in the host language if it is possible to represent the expressions of the application domain as staged data-structures. This technique is called lightweight modular staging (LMS) [32]. LMS interprets domain specific constructs as data-structures. The evaluation of the expression is postponed when using staging.

Based on the staged representation of the code, domain specific optimisation's may be applied. The optimisation techniques are defined by implementing a recursive optimise method for all staged data-structure types. To generate code each staged data-structure type has to implement a compile method. Compile will recursively be called for all staged data-structures and generate code in the target language. When using multi-methods for optimisation there are three main problems which the developer of the multi-stage programming language has to look out for: "separate type-checking/compilation, ensure non-ambiguity, and ensuring exhaustiveness".

Dynamic Staging

Language Virtualization still expose the staged representation to the user. Language Virtualization and LMS use the host language as a meta-generator where the staged representation can be accessed using the staged data-types.

```
def foo(x: Rep[Int]) = {
   val f = (x: Rep[Int]) => foo(x + 1)
   val g = lambda(f)
   g(x)
}
```

Listing 2.1: The staged data-types, here Rep[Int], has to be explicitly stated when using LMS [32]

The host language and the staged programming language cannot be seamlessly combined into a single continuing program. The staged data-types has to be explicitly used. To overcome this, dynamic staging [12] may be used. Dynamic staging completely hides the staging process from the developer by enabling host language constructs to implicitly merge into a staged representation. The staged representation is seamlessly deconstructed and executed when the result is required as a host language construct. This makes the staged DSL more similar to an external DSL, where domain specific optimisation's may be applied to an IR without exposing the IR to the user. Dynamic staging can also combine different dynamic staged DSLs embedded in the same host language into a single program. Merging programs written in different DSLs is a key restriction when using external DSLs. **Not applicable to us. Would require an Scala to Rust interpreter since the program written in the DSL will be both Scala and the staged representation.**

[35] demonstrate a dynamic staged programming language embedded in Java as a library. The programs written using the library are staged implicitly. Data types in the staged programming language are represented as classes in Java. Operation on these data types are implemented as methods in the classes which return a new staged representation. The staged representation will be materialised directly as some operation is applied which will result in a host-language value which cannot be staged. or alternatively when the value goes out of scope.

2.3 The Expression Problem

The negative aspect of deep embedding of DSLs and staged programming languages embedded in a host language is that adding new language constructs will be cumbersome. The new construct's interpretation has to be added to each of the interpreters. Two examples of interpretation are evaluation of an data-structure representing a program written in the DSL and comparison of two DSL programs if they are equal. If a new construct is added to the deep embedded DSL, each interpretation has to be extended to handle the new language construct.

This is weighted against shallow embedding's negative aspect that creating a new interpretation requires a complete re-implementation of the interpreter for all language constructs. Thus, extending the language interpretation requires a lot of work in a shallow embedded DSL. The problem of weighing this two aspect, effective and modular extensibility of language constructs or of interpretations, is called the expression problem [37].

[43] propose two solutions for the expression problem when using deep embedded DSLs. One solution based on object-oriented decomposition which enables easy extension of data-structures in the DSL. Each interpretation of the DSLs data-structure is composed as a function in an interface. All language constructs has to implement the interface and define the interpretation of each function based on the semantic meaning of the data-structure. Therefore, it is easy and modular to add new language constructs to the DSL. Introducing new interpretations means that the interface will be extended with functions, which in turn will require extension of all classes implementing the interface. This means that all language constructs in the DSL has to be extended to implement the new function. The second is a functional decomposition, which favours extension of interpretations. Each interpretation is implemented as a trait in Scala. The trait implement the interpretation for each language construct.

2.4 The Tag Problem

Implementing a typed subject language, DSL or staged programming language, in a typed host language is keen to cause problems [10]. Operators in the DSL should usually be applicable to a variety of types, e.g. a simple addition should work for both doubles and integers in the object language. This is a problem when implementing the representation of the object language addition operation in the host language. A higher order function has to be introduced which can abstract over DSL terms of both double and integer types. But this will result in a loss of type-safety for the DSL. The representation in the host language will be able to mix the concrete types which the abstract representation encapsulates. In the previous example, with the addition of either doubles and integers, the same problem would arise. If a general type was used to abstract both doubles and integers, then a representation of an addition in the DSL may be incorrectly constructed as an addition of an integer and a double. The typed host-language will not catch the erroneous construction since it is type-safe according to the host-language.

2.4.1 Redundant Tagging for Staged Interpreters

[31] argues that a staged interpreter targeting an external language and which use the target language's compiler to compile the generated code has two large benefits. The performance will be enhanced and the semantic soundness of the generated programs will be validated by the compiler. The compiler will first do static semantic validation of the generated program then apply optimisation to the code before emitting a binary. Thus, the staged DSL will not dynamically interpret programs at run-time but instead result in a executable binary. This makes the interpreter of the staged representation similar to a compiler for the DSL.

The drawback of using a typed host language for a typed DSL is that static type-checking in the host language can make the implementation of the DSL complex. When the host language is typed, concrete language types in the DSL needs to be abstracted into higher order abstractions as described earlier. Else an interpreter will not be able to interpret language constructs of varying types. This will however endanger the type-safety of the DSL.

To recapture type-safety, tagging to correct concrete types can instead occur during interpreter run-time e.g. using a match clause. Dynamic tagging has three central problems: the dynamic tagging of the types will have performance penalties, exhaustiveness will be the responsibility of the developer of the match clause, and correct interpretation of each type will be developed within the same function.

The performance penalties is incurred since the tagging is dynamically executed, e.g. with a match clause.

Exhaustiveness means that the developer will be responsible for ensuring that the match clause can handle all possible concrete types. Run-time exception will be thrown if one type is unsupported in the match clause.

Locating the interpretations for all types in a single function makes the development of the interpreter centralised. Forcing multiple developers to work concurrently within a single function will affect the development process negatively.

A strong argument against tagging in the host language when the staged DSL is type-safe is that it is redundant. The interpreted staged representation is already guaranteed to be correctly typed. The only reason for doing tagging in the host language would then be to type-check the interpreter [31]. Static tagging in the host-language will result in static type-checking of the interpreter and dynamic tagging will result in dynamic type-checking of it.

2.5 Rust Programming Language

The trade-off between giving the programmer low level control and being able to guarantee safety properties of the program is a usual problem for general purpose languages. No language has been able to achieve both. It has been a prioritised problem in the programming language research domain [24].

Rust, developed at Mozilla Research, claim to have solved the problem without incurring overhead penalties during run-time [27]. Rust follow C++ and gives zero-cost high-level abstraction. One of Rust main concept to ensure safety properties and avoiding data-races is ownership of resources. Variables can be owned by a restricted set of pointers and special rules are employed for accessing the data. Only one pointer can have the right to modify the data at each time. Several pointer can be granted read access, but not simultaneously as one pointer have writing access.

The ownership concept along with other Rust rules severely restricts the developer. Therefore, Rust had to include a construct for unsafe scopes. In an unsafe scope, Rust will allow actions deemed unsafe by the regular Rust rules such as raw pointer manipulation. The unsafe scope is extensively used, especially in standard library, but it is usually wrapped in a "safe" API [5]. By implying safe, the developer assures that no unsafe or undefined behaviours can be made using the API. The Rust compiler will not be able to statically check this safety, so all safety guarantees have to provided by the developer. Even if the risk of undefined behaviour is restricted to specific part of the program, the unsafe scopes, this will still risk enabling unsafe behaviour at run-time which may cause exceptions.

[24] define a semantic model, called RustBelt, for proving soundness of Rust modules which use the unsafe clause but still claim to expose a safe API. Although Rust concepts mostly ensure soundness of developed programs, Rust itself is still being developed and shaped. Thus, RustBelt will help extend Rusts claim for statically checked safety and support the Rust community by locating bugs.

2.6 Scala Programming Language

2.6.1 Meta-Programming

Compile-time meta-programming with Macros have proven to make the software development more effective. Macros can help to circumvent error prone short-cuts during software development such as code-copying. It can also ease the workload for the developer by generating large amounts of boilerplate-code which is required by the language. Macros was introduced in Scala [7] to to help the developer with similar problems. Scala follow Lisp's hygienic Macro expander, which in turn got the idea from Scheme [16]. A hygienic Macro expander respect scope and previous bindings, same as done during run-time.

Scala also support dynamic reflection during run-time of classes, fields, and types.

2.7 Cross Platform Compiling

Low level virtual machine (LLVM) [26] is an compiler framework which consists of an virtual low-level instruction set. The virtual instruction set capture the primitives commonly used to implementation features in high-level languages. This enables a large set of different high-level languages to target the LLVM byte-code during compilation. The byte-code resembles assembly code by not guaranteeing any type safety or memory safety. LLVM assumes that the high-level programming level will decide to which degree type safety and memory safety should be enforced. The LLVM byte-code is virtual, meaning that it tries to be as platform independent as possible.

LLVM creates an IR from this byte-code and apply safe optimisation techniques to it, thus not altering the semantics of the program. LLVM has the ability to link and merge applications written in different high-level languages to one single LLVM byte-code program. The requirement is that the each of the high-level languages' compiler can compile to LLVM byte-code. LLVM can then apply optimisation across the high-level programming language boundaries.

LLVM also support profile-directed optimisation's at run-time. LLVM can take feedback from the executed binary to find hot-paths. A hot-path represent an execution path which is frequently used during run-time e.g. which branch of an if else statement are mostly used. Using this feedback, the LLVM restructures the instructions to improve run-time performance and re-compiles the optimised program to native-code. This is called just-in-time (JIT) compilation and is a part of other higher-level VMs as well. E.g. the java VM optimize at run-time using profile-directed optimisation.

2.7.1 Rust Compiler and Cargo

The Rust compiler is built on-top of LLVM [17] and uses an external linker which is specific for the host platform [6]. Using LLVM as a back-end enables Rust to in principle be cross compiled to a large set of target platforms [39]. Rust do support a large number of host platforms, each to a varying degree. Some are officially guaranteed to work whilst others depend on community effort to work properly [33]. When cross compiling Rust, the C toolchain and linker has to be explicitly set for the target platform [39]. Apart from the C toolchain, a compiled version of Rusts and C's standard library for the target platform is needed to cross compile Rust programs successfully [15]. The default Rust installer called "rustup" does help with fetching a compiled versions of both Rusts and C's standard library. However, finding the correct cross compiler for linking still remains a problem to be manual solved. It can be a tedious process since *"each combination of host and target platform requires something slightly different"* and to get each combination to work *"typically involves pouring over various blog posts and package installers"* [39].

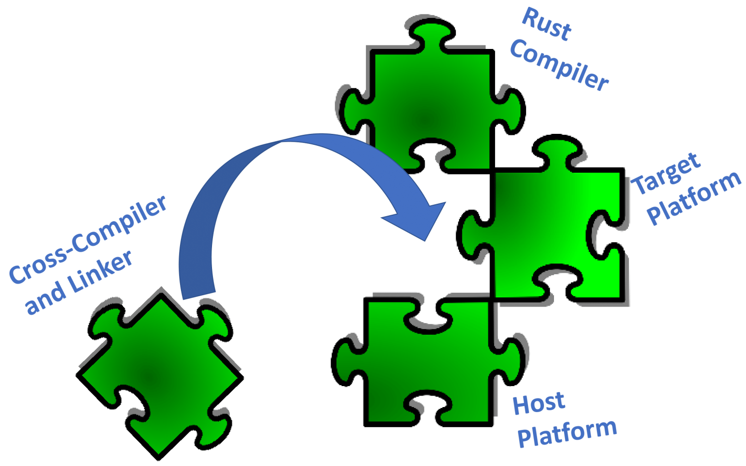

Figure 2.1: Problem while Cross Compiling Rust

The problem can be view as in 2.1. The linker has to be chosen correctly for the specific pair of host platform and target platform. The linker has to be compiled for execution on the host platform but link and emit native code for the target platform. Given that the Rust compiler supports more than 50 different platforms [33], the number of possible host and target platform pairs are theoretically more than 2′500.

2.8 Resource Distribution of the Cluster

Apache YARN [40] is a resource negotiator which decouples the resource management logic from the application logic. YARNs conceptual idea is to act as a distributor of the resources and computing power in a cluster. The resources are distributed in the form of containers. A container is a logical bundle of resources which is used as an execution platform for applications. The YARN architecture is split into three different actors, the Resource Manager (RM), Application Master (AM), and Node Managers (NM).

RM is the central authority of a cluster. All requests for containers has to be accepted by the RM.

Chapter 3

Design

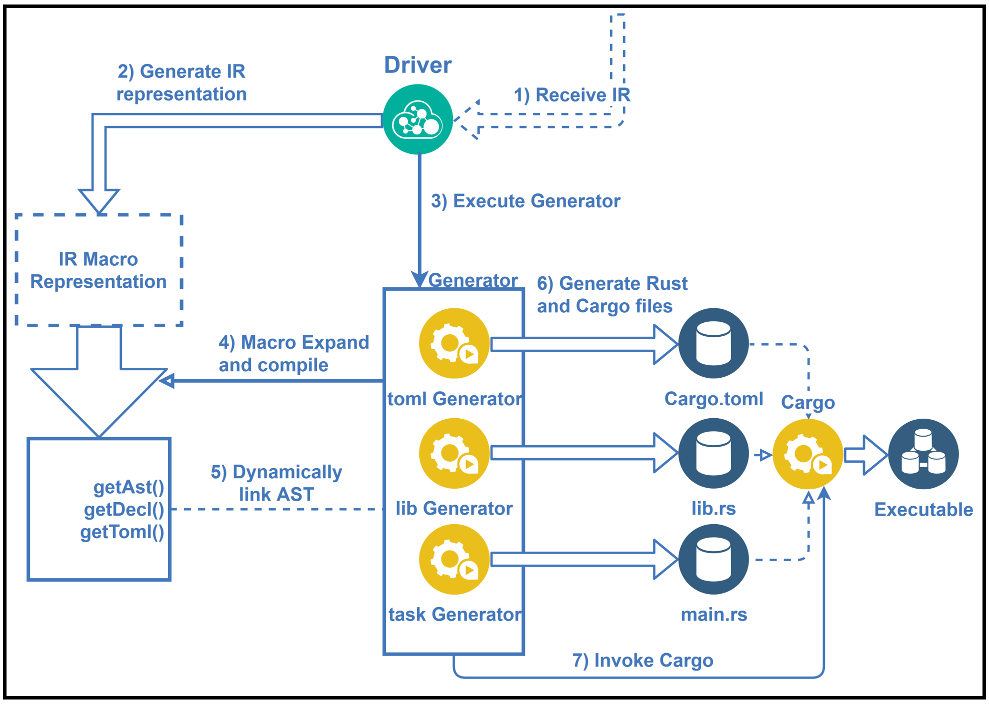

Figure 3.1: Overview of the Rust Code Generator

3.1 Virtual Machine as Host Platform

The Rust project called "cross" aims to supply "Zero setup cross compilation and cross testing of Rust crates" [44]. It depends on docker to run a Ubuntu VM. The VM is initialised to fetch the appropriate linker for the target platform with the Ubuntu VM as host platform. The amount of host and target platform pairs is thus reduced to a bit over 50, which is more manageable compared to the 2'500 pairs without a VM as host platform. cross uses rustup to fetch the correct cross compiled standard libraries.

Chapter 4

Implementation

Chapter 5

Evaluation and Discussion

Bibliography

[1] Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java*. Cambridge: Cambridge University Press, 2002. ISBN: 9780511811432.

[2] Michael Armbrust et al. "Spark SQL: Relational Data Processing in Spark". eng. In: *Proceedings of the 2015 ACM SIGMOD International Conference on management of data*. SIGMOD '15. ACM, May 2015, pp. 1383–1394. ISBN: 9781450327589.

[3] Mikhail Asiatici et al. "Virtualized Execution Runtime for FPGA Accelerators in the Cloud". eng. In: *Access, IEEE* 5 (2017), pp. 1900–1910. ISSN: 2169-3536.

[4] J. Auerbach et al. "Lime: A Java-compatible and synthesizable language for heterogeneous architectures". In: vol. 45. 10. Oct. 2010, pp. 89–108.

[5] Abhiram Balasubramanian et al. "System Programming in Rust: Beyond Safety". eng. In: *ACM SIGOPS Operating Systems Review* 51.1 (Sept. 2017), pp. 94–99. ISSN: 0163-5980.

[6] *Building and Deploying a Rust library on Android*. URL: `https://mozilla.github.io/firefox-browser-architecture/experiments/2017-09-21-rust-on-android.html` (visited on 03/12/2018).

[7] Eugene Burmako. "Scala Macros: Let Our Powers Combine!" In: *SCALA '13 Proceedings of the 4th Workshop on Scala* (2013), p. 3.

[8] Paris Carbone et al. "Apache flink: Stream and batch processing in a single engine". eng. In: *Bulletin Of The Ieee Computer Society Technical Committee On Data Engineering* 36.4 (2015).

[9] P. Carbone et al. "State management in Apache Flink: ® consistent stateful distributed stream processing". In: vol. 10. 12. Association for Computing Machinery, Aug. 2017, pp. 1718–1729.

[10] JACQUES CARETTE, OLEG KISELYOV, and CHUNG-CHIEH SHAN. "Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages". In: *Journal of Functional Programming* 19.5 (Sept. 2009), pp. 509–543. ISSN: 0956-7968.

[11] H. Chafi et al. "Language virtualization for heterogeneous parallel computing". In: vol. 45. 10. Oct. 2010, pp. 835–847.

[12] Piotr Danilewski and Philipp Slusallek. "Building Code with Dynamic Staging". In: (Dec. 2016).

[13] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters". eng. In: *Communications of the ACM* 51.1 (Jan. 2008), pp. 107–113. ISSN: 0001-0782.

[14] Grégory M. Essertel et al. "Flare: Native Compilation for Heterogeneous Workloads in Apache Spark". In: (Mar. 2017).

[15] *Everything you need to know about cross compiling Rust programs!* URL: `https://github.com/japaric/rust-cross` (visited on 03/12/2018).

[16] MATTHEW FLATT et al. "Macros that Work Together: Compile-time bindings, partial expansion, and definition contexts". In: *Journal of Functional Programming* 22.2 (2012), pp. 181–216. DOI: `10\.1017\/S0956796812000093`.

[17] *Frequently Asked Question - The Rust Programming Language*. URL: `https://www.rust-lang.org/en-US/faq.html` (visited on 03/12/2018).

[18] Jeremy Gibbons and Nicolas Wu. "Folding domain-specific languages: deep and shallow embeddings (functional Pearl)". eng. In: *Proceedings of the 19th ACM SIGPLAN international conference on functional programming*. ICFP '14. ACM, Aug. 2014, pp. 339–347. ISBN: 9781450328739.

[19] Benjamin Hindman et al. "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center." In: *NSDI*. Vol. 11. 2011. 2011, pp. 22–22.

[20] P. Hudak. "Modular domain specific languages and tools". eng. In: IEEE Publishing, 1998, pp. 134–142. ISBN: 0-8186-8377-5.

[21] Patrick Hunt et al. "ZooKeeper: Wait-free Coordination for Internet-scale Systems." In: *USENIX annual technical conference*. Vol. 8. 9. Boston, MA, USA. 2010.

[22] Kevin J. Hyoukjoong Lee et al. "Implementing Domain-Specific Languages for Heterogeneous Parallel Computing". eng. In: *Micro, IEEE* 31.5 (Sept. 2011), pp. 42–53. ISSN: 0272-1732.

[23] Yogendra Joshi and Pramod Kumar. "Introduction to Data Center Energy Flow and Thermal Management". In: *Energy Efficient Thermal Management of Data Centers*. Ed. by Yogendra Joshi and Pramod Kumar. Boston, MA: Springer US, 2012, pp. 1–38. ISBN: 978-1-4419-7124-1. DOI: 10.1007/978-1-4419-7124-1_1. URL: https://doi.org/10.1007/978-1-4419-7124-1_1.

[24] Ralf Jung et al. "RustBelt: Securing the foundations of the Rust programming language". In: *Proceedings of the ACM on Programming Languages* 2.POPL (2017), p. 66.

[25] Tomas Kalibera et al. "A fast abstract syntax tree interpreter for R". eng. In: *Proceedings of the 10th ACM SIGPLAN/SIGOPS international conference on virtual execution environments*. VEE '14. ACM, Mar. 2014, pp. 89–102. ISBN: 9781450327640.

[26] C. Lattner and V. Adve. "LLVM: a compilation framework for lifelong program analysis & transformation". eng. In: USA: IEEE, 2004, pp. 75–86. ISBN: 0-7695-2102-9.

[27] Nicholas D. Matsakis and Felix S. Klock. "The rust language". eng. In: *ACM SIGAda Ada Letters* 34.3 (Nov. 2014), pp. 103–104. ISSN: 10943641.

[28] Zubair Nabi, Rohit Wagle, and Eric Bouillet. "The best of two worlds: Integrating IBM InfoSphere Streams with Apache YARN". eng. In: IEEE, Oct. 2014, pp. 47–51. ISBN: 978-1-4799-5666-1.

[29] Eriko Nurvitadhi et al. "Can FPGAs Beat GPUs in Accelerating Next-Generation Deep Neural Networks?" eng. In: *Proceedings of the 2017 ACM/SIGDA International Symposium on field-programmable gate arrays*. FPGA '17. ACM, Feb. 2017, pp. 5–14. ISBN: 9781450343541.

[30] *NVCC :: CUDA Toolkit Documentation*. Mar. 5, 2018. URL: http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html (visited on 03/12/2018).

[31] Emir PaÅ¡aliÄ‡, Walid Taha, and Tim Sheard. "Tagless Staged Interpreters for Typed Languages". eng. In: 2002, pp. 218–229. ISBN: 1-58113-487-8.

[32] T Rompf and M Odersky. "Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs". English. In: *Communications Of The Acm* 55.6 (June 2012), pp. 121–130. ISSN: 0001-0782.

[33] *Rust Platform Support*. URL: https://forge.rust-lang.org/platform-support.html (visited on 03/12/2018).

[34] V. O. Safonov. *Trustworthy compilers*. eng. Quantitative Software Engineering Series. Hoboken, N.J.: Wiley, 2010. ISBN: 1-282-55146-9.

[35] Maximilian Scherr and Shigeru Chiba. "Almost first-class language embedding: taming staged embedded DSLs". eng. In: *Proceedings of the 2015 ACM SIGPLAN International Conference on generative programming: concepts and experiences*. GPCE 2015. ACM, Oct. 2015, pp. 21–30. ISBN: 9781450336871.

[36] Vytautas Štuikys and Robertas Damaševičius. *Meta-Programming and Model-Driven Meta-Program Development: Principles, Processes and Techniques*. eng. Vol. 5. Advanced Information and Knowledge Processing. London: Springer London, 2013. ISBN: 978-1-4471-4125-9.

[37] Josef Svenningsson and Emil Axelsson. "Combining Deep and Shallow Embedding for EDSL". eng. In: vol. 7829. 2013, pp. 21–36.

[38] Walid Taha and Tim Sheard. "MetaML and multi-stage programming with explicit annotations". eng. In: *Theoretical Computer Science* 248.1 (2000), pp. 211–242. ISSN: 0304-3975.

[39] *Taking Rust everywhere with rustup*. May 13, 2016. URL: https://blog.rust-lang.org/2016/05/13/rustup.html (visited on 03/12/2018).

[40] Vinod Vavilapalli et al. "Apache Hadoop YARN: yet another resource negotiator". eng. In: *Proceedings of the 4th annual Symposium on cloud computing*. SOCC '13. ACM, Oct. 2013, pp. 1–16. ISBN: 9781450324281.

[41] Michael Vögler et al. "Ahab: A cloud-based distributed big data analytics framework for the Internet of Things". In: *Software: Practice and Experience* 47.3 (Mar. 2017), pp. 443–454. ISSN: 0038-0644.

[42] Matei Zaharia et al. "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*. USENIX Association. 2012, pp. 2–2.

[43] Matthias Zenger and Martin Odersky. *Independently Extensible Solutions to the Expression Problem*. eng. 2004.

[44] *"Zero setup" cross compilation and "cross testing" of Rust crates*. URL: https://github.com/japaric/cross (visited on 03/12/2018).

Appendix A

Appended Material