**Main Idea:**

Whole system:

DSL to physical IR (not part of master thesis’), physical IR executable code

Physical IR to rust to LLVM IR:

Front-end: physical IR is interpreted to rust AST (Klas)

Back-end: convert rust AST to LLVM IR (me)

Correct AST, using traits and generic types, representation in Scala will ensure static type-checking by Scala of Rust types. **Type checking combined with cross-stage persistence of Rust variables will be non-trivial to implement statically Scala. How to statically type-check usage of assigned variables? E.g. new Let(new Var[IntT](“x”), new Integer(5)) followed by new Add(new Var[DoubleT](“x”), new Double(1.0)) should invalidate type of “x”. May be better us the rust compiler to find these issues.**

**Reason for using labels, let statements and var expressions for variable assignment, access and mutation is:**

**Static type-checking will improve: if only label of the variable would be enough for representing the value assigned to the variable, the static type-checking of expressions would not be able to work, e.g. Add(“x”, Int[IntT](1)) will be hard to type-check, will have to pass all expressions as any type. Add(Var[IntT](“x”), Integer[IntT](1)) would on the other hand preform static type-checking in Scala.**

**Reason for using Let(“x”, Expr[T]) is to avoid breaking cross-stage safety and to avoid ambiguity about reassigning variables available in wider scopes or if it is an assignment of a new variable.**

**If Var would be used with VarMut, the reassignment of variables could also be statically checked by Scala, ensuring that the IR to AST interpreter does not try to reassign an immutable variable since this breaks Rust semantics.**

Traits:

TRustAST – define abstract method generate

TExp[T] – extends TRustAST and implements method generate – define abstract method generate\_expression and generic type RT (return type of Expression, used for static type-checking) – TExp may even extend TStatement to implement generate\_statement as generate\_expression(); emit(“;”)

TStatement - extends TRustAST and implements method generate – define abstract mehod generate\_statement

This will enable classes to safely extend both TExp and TStatmen, so a statement sequence may hold both expressions and statements.

Problem with compiling to LLVM is that NVVM is a subset of LLVMs instruction. Any given NVVM byte-code will be a valid LLVM byte-code but not the other way around. Therefore, the compiled LLVM code may not be able to target GPUs. It may be possible to construct dynamic C and CUDA libraries (.so) and link them dynamically along with the rest of the run-time tools.

On the upside, run-time profile based optimizations will be possible by the workers, which means that they can reoptimize and recompile the code to take advantage of hot paths.

**Proposal:**

Take RustAST and compose to rust code. Use rust compiler (optionally Cargo) to generate LLVM IR code for specific architectures. Dynamically link runtime tools e.g. CUDA.

**Possible solution:**

Compose an interpreter in Scala to compile the AST to semantically equal rust code. Do not check or guarantee correct rust code will be omitted. If the ast code represent invalid rust semantics, e.g. it breaks ownership rules, the rust compiler will catch that and report it back. The ast interpreter does not check for semantically correct rust ast.

**Possible evaluation criteria:**

How should error messages be passed in IR to LLVM IR code interpretation? All compile errors will occur during the compilation with rustc/cargo, but the problem can have several origins. Either the rust AST to LLVM IR transformation is incorrect in some fashion, creating LLVM IR/rust code which differ semantically from the rust AST. Or, the interpreter from physical IR to rust AST creates a AST which is semantically incorrect according to rust, e.g. breaks ownership rules. A third error could be exception in dynamically linked libraries, which could be the responsibility of either part of the physical IR to LLVM IR interpreter or a bug in the dynamically linked library.

**General continuous big data analytics background**

**Spark**

*Source: Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing*

Spark is an expressive distributed computing framework running on the Java virtual-machine (VM). Spark aims to improved performance by avoiding moving data eagerly to external stable storages. Other popular distributed computing frameworks prior to Spark moved intermediate result to an external stable storage, 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 are abstract execution plans on data stored in stable storage. RDDs are lazily evaluated. No intermediate results are calculated before a result value from the expression the RDD represents is requested. The only exception to this is if the user explicitly instructs the intermediate result to be persisted to stable storage. At the time when the result value is requested, Spark will optimize the execution plan and distribute the workload over the cluster. The effects Spark had on performance was an improvement of great magnitude, up to 20x compared to Hadoop.

**Spark SQL**

*Source: Spark SQL: Relational Data Processing in Spark*

Application analyzing Big Data tend to utilize 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 optimization to match the performance of frameworks specialize for relational queries. To tackle this problem, Spark introduced the DataFrame concept along with an optimizer specifically developed for query optimization called Catalyst. DataFrames represents a distributed set of rows with a uniform schema. These rows can be manipulated using the DataFrame API, which consists of a set of lazily evaluated relational operators, similar to RDD transformations. A DataFrame can be transformed to an RDD of row objects. This bridge the gap between relational queries and procedural algorithms.

**Flare**

*Source: Flare: Native Compilation for Heterogeneous Workloads in Apache Spark*

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. Operations on a DataFrame will originally be executed on the Java VM. Running operations in a VM is inherently slower than native-code. Therefore, Flare transforms SQL queries on DataFrames to native code and improves performance significantly. Sparks query performance with Flare matched the performance of the best SQL engines without affecting Sparks expansive expressiveness. **(CDA is an attempt to make a complete system compile to native code, or at least as much as possible. Performance can probably be significantly improved doing this.)**

**Flink**

*Sources: State Management in Apache Flink*

*Apache flink: Stream and batch processing in a single engine*

*https://ci.apache.org/projects/flink/flink-docs-release-1.2/concepts/runtime.html*

Flink is an open source system focused on continuous distributed computing. Flink’s system will be scheduled once and long running. Therefore, it needs to be adaptive and fault tolerant. To achieve this, Flink employs a dynamic cluster architecture which is composed of three processes: the client, the Job Manager and the Task Managers.

The client will compile and optimize the logical pipeline before sending it to the Job Manager. The Job Manager is responsible for coordination, physical deployment and fault tolerance. The system can even handle Job Manger failures using leader election in ZooKeeper. The Task Managers are workers, using a single Java VM to execute tasks assigned to it.

The data-transmissions are performed in two ways, pipelined or blocking data exchange. Pipeline is the preferred method, since it enables concurrent execution of producer and consumer. When using pipeline, the consumer will continuously transmit partial results to the consumer during its execution. This reduce the memory requirements for buffers and also allows backpropagation of pressure, i.e. the producer will be notified if they produce results to slow.

During block data exchange, the producer will not send any intermediate results until all it’s input has been consumed. This separates the execution of producer and consumer into stages, and has a higher probability to spill to secondary storage due to larger memory requirements for buffers.

**Weld**

*Source: Weld Rethinking the Interface Between Data-Intensive Libraries*

Performance of data-intensive computations suffer heavily from data movement due to usage of different libraries within the same program. This is caused by the fact that compilers cannot optimize across library boundaries. Instead all intermediate results are forced to be calculated and stored in memory before passing control to an external library. Weld has shown that huge speedups in execution time can be achieved by addressing this problem. Weld implement a cross-library optimizer to enhance performance of data-intensive computations on bounded data. To enable the cross-library optimizations, each library must supply an IR representation of their program using Welds’ IR representation. This IR can be generated after the source code has been optimized to achieve domain specific optimizations, such as rearrangements of linear algebra operations. Weld then constructs a complete IR of the program containing all function calls across libraries. Then Weld optimizes the whole IR tree. The final resulting IR tree is then used to compile to hardware specific code.

Weld’s compile time, including optimization of the IR, ranged from 62-256ms. The mean compile time was 126ms and median 117ms. Since the target users will apply weld to programs analyzing huge datasets this compile time seems acceptable. The compilation time would be amortized constant time if Weld were applied to a long running stream processor.

**DSL background**

**Deep Embedding of DSLs**

*Source: Modular Domain Specific Languages and Tools*

General-Purpose Languages are great tools for programmers who’s work spans over several application domains. But a General-Purpose Language might imped the development process for user which has limited experience with software development and who are specialized in a specific application domain. A more precise and narrow language may prove beneficial for such developers. Such languages are called domain specific language (DSL). A DSL is specifically developed for a target application domain. 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, most semantics, development tools and other related artifacts is reused by the DSL. This will reduce development cost of the DSL significantly and make it less error prone. A pure embedding will leave the host language constructs, e.g. multiplication using the \* operator, unaltered.

Focus during the development of the DSL can instead be shifted from syntax to the semantics of the domain specific parts of the DSL. These semantics should be clear and capture the intuition behind the domain concepts to simplify the formal process of proving the correctness of a program.

**Deep vs shallow embedding**

*Source: Folding Domain-Specific Languages: Deep and Shallow Embeddings*

*Source 2: A Fast Abstract Syntax Tree Interpreter for R*

Embedding of DSL is widely split into two categories, shallow and deep embedding.

Shallow embedding is done by implementing the DSL constructs as constructs directly in the host language. The DSL constructs will then be compiled and executed along with the rest of the host language. Even though interpreting and executing instruction directly, as virtualized byte-code, domain specific optimization can mitigate the over-head of AST creation and improve performance. **[Source 2]** 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 AST structure, domain specific optimizations 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. The main reason for using Java was for simplicity of the java VM, taking advantage of garbage collection, dynamic loading of code and type-safety of a object-oriented language. These benefits are gained at the expense of run-time performance. The argument for this is that using java, the developer of the interpreter do not have to have deep knowledge of the compilers. This is usually not the case in domain specific, but java will enable experts in the application domain to extend the interpreter with domain specific optimization techniques.

Shallow embedded DSLs require a partial evaluator to apply domain specific optimizations to the generated code. Implementation of a partial evaluator requires a substantial effort. Therefore, deep embedding avoids involving a partial evaluator at all for domain specific optimization 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 optimizations can be applied directly do it. The resulting, optimized IR is then interpreted to corresponding constructs in the host language. The whole application is finally compiled using the host language compiler.

**The Expression Problem**

*Source 1: Combining Deep and Shallow embedding for EDSL*

*Source 2: Independently Extensible Solutions to the Expression Problem*

The negative aspect of deep embedding of DSL is that adding a new language construct will be hard. 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 embeddings negative aspect that creating a new interpretation requires a complete reimplementation 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. **[Source 1]**

**[Source 2]** 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 favors extension of interpretations. Each interpretation is implemented as a trait in Scala. The trait implement the interpretation for each language construct.

**Modular Development of DSLs in Scala Using Traits**

*Source 1: Language components for modular DSLs using traits*

*Check sources [28]*  ***Scala traits and how to resolve conflicting implementations***

Developing a DSL is usually an iterative process. Divide the DSL into components can enable more modular and therefore parallel development. One technique for achieving this is utilizing traits, which is similar to inheritance. A trait can be used similarly to an abstract class, encapsulating a number of different concrete classes which is guaranteed to implement the trait’s methods.

Interpretation of an AST from the AST can easily become centralized. If a function is used to parse the AST, taking as input a instance, the function become responsible for implementing the interpretation of the class. It also has to assert that it is exhaustive for all possible concrete classes which it may receive as input. A more modular approach, when using an object-oriented language, is to define an abstract method in a trait. Thus, the concrete classes and objects will be abstracted into a higher-order. The responsibility of exhaustiveness and function calling is then be shifted to the polymorphism of the host language. The actual implementation of the method will be the responsibility of the developer of the concrete class implementing the trait. Therefore, adding new nodes to the AST will be modular and less error prone.

Traits may also be used to enforce static type-checking of the AST using generic types. Traits may be used to represent higher-order abstract types, such as an expressions. If language interpretation uses the traits instead of concrete classes, or at least has a default interpretation of the general case for a generic type implementing a trait, new language constructs can easily be added to the deep embedded DSL. The new constructs only have to extend the appropriate traits and implement the trait’s required functions. Interpretation by other language constructs will then be automatically handled by polymorphism in the host language.

**Language Virtualization**

*Source: Language Virtualization for Heterogeneous Parallel Computing*

Modern software development, especially in the Big Data domain, become more focused on scale-out performance. This means that instead of improving the capacity of a single computer, scale-up, the program will run on a large set of possibly distributed nodes. Coordination in a distributed setting is non-trivial and error prone. Therefore, most developers turn to abstract models for parallel programming where the distribution is implemented implicitly by the framework **(references to Spark/Hadoop etc)**. This will work quite good on homogenous cluster, e.g. Flink and Spark which run on java VMs, but most frameworks for distributed computing does not fully utilize the capability of a heterogenous cluster.

To fully utilize the potential of such a cluster the emitted code would need to be specifically tailored for the underlying hardware of each individual node. This is a real problem since vendors of accelerators, e.g. GPUs, tend to update and release new APIs continuously which makes more efficient use of their hardware. If an application used these APIs directly it would not be portable to new hardware. To make such an application portable to new hardware would require substantial efforts to continuously reimplement the application.

Language Virtualization is a method to alleviate the application developer of the problem of continuous portability updates whilst optimizing the usage of the underlying heterogenous cluster. The multi-staged programming language application will first be transformed to an intermediate representation (IR). To avoid giving the application developer the feeling of writing a code-generator and guarantee semantic safety, several concepts in the host language is virtualized by overloading them. Overloading a concept means that the actual computation which the concept expresses in the host language will not be executed. Instead, the concept will be turned into a corresponding representation in the IR. Language constructs of the host language which are not a part of the multi-staged programing language can not be exposed to the application developer.

The reason the program and the overloaded constructs should be turned into a IR instead of interpreting them directly to the target language is to enable domain specific optimizations. The IR can be optimized aggressively using domain specific knowledge. The same optimizations without the IR would require a substantial effort using a partial evaluator and would most likely not achieve the same degree of optimization.

The optimized IR is used to generate code in the target language. In a heterogenous cluster, the IR can be used to generate code in several different target languages. Thus, the multi-staged programming language will be a valid abstraction which still can take full advantage of the underlying hardware.

**Multi-stage (staging) programming**

*Source: MetaML and multi-stage programming with explicit annotations*

*Check sources [7* ***(Runtime code generation)****, 20 (partial evaluation), 47]*

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, same as embedded DSLs. 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 use the host language as a meta code generator and exposes staged representations to the DSL 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 tradeoff for embedded DSL is that the user may find it hard to understand what the staged representation of the program will actually look like.

The staged AST representation enables validity checks of the semantics to verify it’s correctness, as type-checking the AST. The AST can be optimized using domain specific knowledge. Source code in the target language is generated from the AST representation by the generator. The AST representation of the staged program have four key properties:

Staging annotations – the DSLs constructs to build and manipulate the staged representation.

Static type-checking – using generic types and traits, the staged representation can be statically type-checked by the host language.

Cross-stage persistence – variable assignments of variables will be available in later stages. Similar to regular variable assignment in scopes.

Cross-stage safety – unbound variables cannot be used in stages unless they are assigned in lower stages. Similar to cross-stage persistence but a safety aspect of it.

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 general purpose languages are composed of two stages; compile-time and run-time.

The generation stage will interpret the DSL constructs, stage them and compile to target source code. 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.

**Lightweight Modular Staging**

*Source: Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs*

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). 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 optimizations may be applied. The optimization techniques are defined by implementing a recursive optimize 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 optimization there are three main problems which the developer of the multi-stage programming language has to look out for:

“separate type checking/compilation, ensure nonambiguity, and ensuring exhaustiveness”.

Static type-checking by the host language should be used to simplify debugging, instead of using compilation errors from the target language’s compiler to report semantic errors and propagate it back to the stage which caused them.

Nonambiguity should be ensured by having a clear structure of the staged representation where the semantics cannot be ambiguous to the developer. Following implementation rules where the narrowest most specific implementation of traits will be used will ensure nonambiguity. Each staged type will only use one implementation of each trait no matter the surrounding environment.

By having default implementation of traits with generic types, exhaustiveness can be guaranteed for specific type. Either a concrete type has a specific implementation of the trait or it will use the default implementation on the generic type.

**DSLs for distributed computing**

*Source: Lime: a Java-Compatible and Synthesizable Language for Heterogeneous Architectures*

Lime is another implemented language virtualization method. It introduces a DSL within Java with a set of language constructs which limits the usage of global fields or none final static fields and encourage 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. Global methods may not override or implement local methods but local can override and implement global functions. 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 synchronization. These tasks can therefore be dynamically allocated to hardware components in the cluster. No distributed coordination aside from input-output pipelining need to be emplaced 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 more evenly over the cluster. A running graph may be dynamically extended with new task. Added task will automatically be connected and started. Explicit loops in the dataflow-graph are illegal, such a graph will throw an exception when run. Feedback is instead possible through messaging techniques between tasks.

**Ziria**

The implementation of wireless protocol is none trivial 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 tailormade for the underlying hardware, but it makes the software development process more complex and error prone. Ziria is a DSL which has been implemented to enhance development and maintenance of wireless systems. It consists of two main components: stream transformer and stream computer.

The transformer takes an input stream, transform the data, and passes the transformed data to an output stream. The computer is similar, but besides consuming the input and emitting output, it produces another value called control value. The workflow of a stream computer is as follows, it continuously consumes input and produces output which is passed downstream. After a while the computer halts and emits the control value. The workflow of the pipeline is not delayed by the computers since they continuously emit output when calculating the control value. The control value can be metadata about the input stream. Ziria can use this to reconfigure the whole pipeline downstream. An example from the paper is a payload header decoder, where the computer emits a control value corresponding to which decoder should be used for the payload body. The rest of the downstream pipeline can then dynamically be initialized as the required decoder for the current input in the stream.

**IR and target code background**

**Thorin, an IR for imperative and functional languages**

Thorin is a IR following the continuation-passing style (CPS) which is common to functional languages. One issue with CPS is the access of variables using names, e.g. where the semantics of a closure refers to the wrong variable definition when intending to be a free variable. Therefore, Thorin does not support accessing variables by labels in the IR. This solves some scoping ambiguity issues with variable name capturing. The IR is instead transformed to a graph, where usage of variables are indicated by edges, i.e. references, to the nodes defining the variables in the graph. Functions calls do not return any values, their IR graph will instead be in-lined to, i.e. merged with, the programs complete IR graph. Since variables and functions cannot be accessed using labels, Thorin do not need to support any scope nesting. Thorin argues that any imperative program using scope nesting can be transformed to a CPS IR without scope nesting. Scopes are used in imperative languages to create an unambiguous semantics for usage of variables and enabling re-usage of variable names, called shadowing of variables, but they are omitted once the AST is created. **(But they will be necessary for generating Rust code, since Rust’s ownership rules makes a program reliant on correct scoping for destruction of variable borrows and references. Removing labeling of variables in the RustAST may be interesting but labels will have to be supported for using e.g. external functions).**

**LLVM, an opensource compiler framework**

Low level virtual machine (LLVM) is an compiler framework which consists of an virtual low-level instruction set which capture the primitives commonly used to implementation features in high-level languages. This enables a large set of different high-level languages to target LLVM byte-code during compilation. The byte-code resembles the assembly code of the machine by not guaranteeing 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.

LLVM creates an IR from this byte-code and apply safe optimization 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 optimization across the high-level programming language boundaries which existed the original applications.

LLVM also support profile-directed optimizations 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 followed e.g. which loops or branches of if else statements in the LLVM representation are mostly used. Using this feedback, the LLVM reorganize the instructions to improve run-time performance and recompiles the optimized 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 optimizations.

**Compiling a larger set of general purpose languages to JVM through LLVM IR**

*Source: Sulong - Execution of LLVM-Based Languages on the JVM*

The java VM have in recent years been popularized when it comes to hosting other languages than java. The large benefit being that the language only need to implement one target binary code, the java bytecode. The binary executable can be ship to any platform which has the java VM installed, no platform specific compilation are needed.

Sulong is an attempt to bring a large set of programing languages to the java VM by using LLVM IR as an intermediate step between source code and java bytecode. The LLVM IR is interpreted by the Sulong interpreter into an AST which subsequently is transformed to java byte-code. Sulong has some limitations, in the paper one example is the transformations of pointers in C by converting them back and forth to integers and applying integer transformations on them. This is undefined behavior, but the static compiler, C compiler, behaves in a way which enables the developer to create a program which achieves the developers intended behavior. Sulong does not aim at supporting this, therefore it will not be able to support all real-world applications.

**Rust’s foundation for statically checked correctness of programs**

*Source 1: RustBelt Securing the Foundations of the Rust Programming Language*

*Source: The Rust Language*

The tradeoff 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 supply both. It has been a prioritized problem in the programming language research domain.

Rust, developed at Mozilla Research, claim to have solved the problem without incurring overhead penalties during run-time. 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 severely restricts Rust. Therefore, the unsafe scope construct is also a part of rust. In an unsafe scope, Rust will allow actions deemed unsafe by the regular Rust ownership 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. By the safe API, the developer assure that the library will not have any unsafe or undefined behaviors. The Rust compiler will not be able to check this safety, so all safety guarantees have to provided by the developer. Even if the risk of undefined behavior is restricted to specific part of the program, the unsafe scopes, this will still open up for unsafe behavior equivalent to C and C++.

**[Source 1]** 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.

**Code-generation for heterogenous platforms background**

**Future of heterogenous clusters**

*Source: Can FPGAs Beat GPUs in Accelerating Next-Generation Deep Neural Networks?*

Deep Neural Networks (DNNs) depend on effective computations of floating point operations. Therefore, GPUs are usually utilized for their superior parallel computing power. An architecture which has an emerging potential for substituting and complementing GPUs are Field-Programmable Gate Array (FPGA). The main advantage of FPGA is their energy efficiency. [Source] compare the performance and energy efficiency of future FPGAs compared to current state-of-art GPUs. The FPGA was able to perform 10 to 60 better than the GPU whilst requiring about half the energy consumption per operation. This shows that the state-of-art heterogenous platform have a probability of changing in the future. Therefore, distributed computational frameworks should be easily extendable to new hardware platforms.

**Code-generation of effective code for distributed-memory architectures**

*Source: Code Generation for Distributed-Memory Architectures*

Most frameworks for code-generation targeting distributed-memory architectures separate the code-generation from the dataflow optimizations. [Source] propose to combine these, and wait for compiler optimizations, such as loop fusion, before deciding where to transfer data.