A Scala DSL for Rust code generation

Klas Segeljakt <klasseg@kth.se>

The Royal Institute of Technology (KTH)

30 maj 2018

Abstract

Continuous Deep Analytics is a new type of analytics with performance requirements exceeding what the current generation of distributed systems can offer. The backbone of general purpose data processing systems such as Spark and Flink is the Java Virtual Machine (JVM). While the JVM is portable, it is a also performance bottleneck due to its garbage collector. A portable runtime with high-performance, capable of running efficiently on heterogeneous architectures, is needed. We have developed a code generator as a framework for this runtime. Instead of running portable but slow Java code, optimized Rust code will be generated and cross compiled to different architectures. By generating Rust code, as opposed to C or C++ code which is the common approach, we achieve stronger safety guarantees. The code generator will be used in a five year project by KTH and RISE SICS. In this project a new distributed system will be developed to meet the demands of CDA. Due to time constraints, the generator is at present restricted to a sample of the Rust language. It is however designed to be extensible in respect to adding new constructs. Adding full support for all of Rust's features is left for future work.

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

Sammanfattning

N/A

Contents

List of Tables

List of Figures

1 Acronyms

Acronym	Expansion
CDA	Continuous Deep Analytics
AST	Abstract Syntax Tree
IR	Intermediate Representation
DSL	Domain Specific Language
GPL	General Purpose Language
JVM	Java Virtual Machine
UDF	User Defined Function
FOAS	First Order Abstract Syntax
HOAS	Higher Order Abstract Syntax
SFI	Software Fault Isolation
IFC	Static Information Control

The Rust Programming Language

C and C++ have for many decades been the sole low level system development languages [1]. While both are very optimized, they are also unsafe. Mistakes in pointer aliasing, pointer arithmetic and type casting can be hard to detect, even for advanced software verification tools. Although C++ facilitates countermeasures for these issues, e.g., smart pointers, RAII, and move semantics, its type system is too weak to statically enforce their usage [2]. 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 [2]. 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 affords many zero-cost abstractions, e.g., pattern matching, generics, traits, and type inference.

Packages, e.g., binaries and libraries, in Rust are referred to as crates. Cargo is a crate manager for Rust which can download, compile, and publish crates. A large collection of open-source crates can be browsed on https://www.crates.io. Rust has a stable, beta, and nightly build. To date, the largest crate is the Servo browser engine. 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 [3], and a runtime with green threaded-abstractions [4].

1.1 Basics

A Rust crate is a hierarchy of modules. Modules contain structs, traits, methods, enums, etc., collectively referred to as items. Structs encase related values, and do not support inheritance. Listing X defines a struct Rectangle and a tuple struct and Triangle. Tuple structs are a variation of structs where fields are unnamed.

Listing caption 1: #rust1

```
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 **???** wraps values of type Rectangle and Circle.

Listing 1: Enums.

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

Traits define methods for an abstract type Self and can be implemented in ad-hoc fashion, comparable to type classes in other programming languages. In listing **???**, Geometry is a trait which defines a method for calculating the perimeter. Rectangle, Triangle and Shape implement the Geometry trait.

Listing 2: Traits 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. Finally, listing **???** initializes a Rectangle and calculates the perimeter.

Listing 3: Struct initialization and method invocation.

```
fn main() {
  let r = Rectangle {
    width: 3,
    height: 4
  };
  println!("{}", r.perimeter()); // 14
}
```

1.2 Syntax

Rust's syntax is mainly composed of *expressions* and *statements* [5]. 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 indexing, field access, or dereferencing operation, and can be assigned to if mutable. Value expressions represent a value, e.g., a binary operation or function call, 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 ignore the result, and in consequence only produce side-effects. Listing X displays examples of various statements and expressions.

Listing 4: Examples of syntactic units from different categories.

```
struct Foo;         // Item declaration
```

```
let foo = Foo;      // Let declaration
loop { break; }     // Expression statement
loop { break 5; } // Value expression
(1+2)               // Value expression
*(&mut (1+2))       // Place expression
foo                 // Place expression
```

1.3 Ownership

When a variable in Rust is bound to a resource, it takes *ownership* of that resource [1]. 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 [6].

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 have overall been restrictive or required verbose annotations [2]. In addition, ownership is able to solve complex security concerns such as Software Fault Isolation (SFI) and Static Information Control (IFC) [1].

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.

1.4 Lifetimes

Every resource and reference has a lifetime which corresponds to the time when it can be used. 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'.

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 **???** displays how Rust is able to infer the type of a variable based on information past its declaration site.

Listing 5: Type inference example.

```
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 **???**. 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 6: Type annotations, lifetime annotations, and lifetime elision.

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

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

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

1.5 Subtyping, Variance, and Coercions

Rust supports subtyping of lifetimes, but not structs [7]. 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 also includes type constructors. A type constructor is a type which takes type parameters as input and returns a type as output, e.g., Option<T> or &'a mut T. 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.

1.6 Unsafe

Ownership and borrowing rules can in some cases be too restrictive, specifically when trying to implement cyclic data structures. 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. 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. RustBelt is an extension to Rust which verifies the soundness of unsafe blocks. 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.

1.7 Compiler overview

Rust's primary compiler is *rustc* [8]. A general overview of the stages *rustc* takes to compile source code into machine code are illustrated in figure **???**.

Lexing Rust's lexer distinguishes itself from other lexers in how its output stream of tokens is not flat, but nested. Separators, e.g., paired parentheses, braces, and brackets, form token trees which are later dispatched to the macro system. As a side effect, mismatched separators are among the first errors detected by the front-end. The lexer will also scan for raw string literals [9]. 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 more flexibility. A non-canonical grammar for Rust is available in the repository [10]. 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++ [11]. JFlex is however a close alternative to Flex which targets Java [13]. 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. 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. Other are reserved for later stages of compilation. A conditional compilation attribute 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*.

Name resolution Macro expansion and configuration is followed by name resolution. 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). 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 based on the Hindley-Milner (HM) algorithm, with extensions to enable subtyping, region inference, and higher-ranked types. As input, the HM algorithm takes a set of inference variables, also called existential variables, and unification constraints [14]. 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. [15] 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. The Hindley-Milner used in Rust is local to function bodies. 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. 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 variables are inferred lazily, 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 all 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.

Trait resolution During trait resolution, references to traits are paired with their implementation. 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. 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. All types are explicit and

Borrow Checking

1.8 Non-lexical Lifetimes

Lifetimes in Rust are currently lexical. Thereby, a lifetime is always bound to some lexical scope. This model will be changed in the near future to non-lexical lifetimes which allow for more fine-grained control [17]. Non-lexical lifetimes are resolved through liveness analysis. A non-lexical lifetime ends when its value or reference is no longer live, i.e., when it will no longer be used at a later time. It is possible to determine a lexical lifetime solely by analyzing the syntax tree. Non-lexical lifetimes require more information, and are as a result calculated at a later stage when the compiler has assembled the control-flow graph. A comparison, highlighting one advantage of non-lexical lifetimes can be viewed in Listings X and Y.

Listing 7: Lexical lifetimes. Both a and b's lifetimes ends when going out of scope. Since their lifetimes overlap, a.push(9) results in an error.

```
  fn main() {                 //   'a
    let mut a = vec![];       // <-+
    a.push(1);                //   |
    a.push(2);                //
|'b
    let b = &a;
// <-+-+
    foo(a);                   //
| |
    a.push(9); // ERROR       //
| |
  }
// <-+-+
```

Listing 8: Non-lexical lifetimes. No error occurs since b's lifetime ends earlier, i.e., when it will no longer be used.

```
  fn main() {                 //   'a
    let mut a = vec![];       // <-+
    a.push(1);                //   |
    a.push(2);                //
|'b
    let b = &a;
// <-+-+
    foo(b);
// <-+-+
    a.push(9); // OK          //   |
  }                           // <-+
```

‘‘‘

Ownership is enforced by the Borrow Checker. The soundness of the Borrow Checker has formally been proven to be correct in a research project named Patina [18].

https://internals.rust-lang.org/t/possible-alternative-compiler-backend-cretonne/4275/41
https://users.rust-lang.org/t/proposal-rllvm-rust-implementation-of-llvm-aka-just-use-cretonne/16021/1

https://blog.rust-lang.org/2016/04/19/MIR.html

https://github.com/Cretonne/cretonne

[1] 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.

[2] 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.

[3] T. R. 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]

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

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

[6] 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.

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

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

[9] T. R. 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]

[10] T. R. Team, "Rust - canonical grammar," 2015. [Online]. Available: https://github.com/nagisa/rfcs/blob/grammar/text/0000-grammar-is-canonical.md. [Accessed: 17-Apr-2018]

[11] G. O. S. The Free Software Foundation, "Bison," 2014. [Online]. Available: https://www.gnu.org/software/bison. [Accessed: 17-Apr-2018]

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

[13] J. Team, "JFlex," 2018. [Online]. Available: https://github.com/jflex-de/jflex. [Accessed: 17-Apr-2018]

[14] 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]

[15] P. VAN WEERT, "EXTENSION and optimising compilation of constraint handling rules," 2010.

[16] M. Carro, "Constraint logic programming," 1987.

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

[18] E. Reed, "Patina: A formalization of the rust programming language," *University of Washington, Department of Computer Science and Engineering, Tech. Rep. UW-CSE-15-03-02*, 2015.