Bachelor's Thesis

Autos: A Self-Contained, Self-Compiling Rust Compiler Targeting a Self-Emulating LLVM IR Emulator

Author

Sam Crawford

Supervisor

Univ.-Prof. Dipl.-Inform. Dr.-Ing. Christoph Kirsch

submitted in partial fulfillment of the requirements
for the degree of Bachelor of Science in Informatics

Paris Lodron University of Salzburg

Department of Computer Science

26th October 2026

Abstract

This thesis investigates how a self-contained, self-referential software system can be built to compile and emulate itself. The Selfie system demonstrates this concept using C*, a subset of C, as its implementation language and RISC-U, a subset of the RISC-V instruction set architecture, as its target language. Since Selfie is written in C*, it can compile its own source code into RISC-U, which its emulator can then execute, producing a program that can compile itself again and, in principle, repeat the process indefinitely.

Inspired by Selfie, this thesis presents Autos, a self-contained system that follows the same principle using Rust and LLVM Intermediate Representation (LLVM IR) as opposed to C and RISC-V. The central challenge lies in defining suitable subsets of both languages that enable both self-compilation and self-emulation while preserving the essential characteristics of Rust and LLVM IR. An additional challenge was resolving unexpected performance issues, which, although not a primary focus of this work, were largely addressed by revisiting the design of those subsets. To this end, this thesis introduces *Rust, a subset of Rust, and LLLVM, a subset of LLVM IR. Like Selfie, Autos is implemented entirely in its source language, *Rust, allowing it to compile its own source code into LLLVM and execute the resulting program through emulation. This demonstrates how carefully designed language subsets can enable self-reference in a self-contained software system.

Contents

1 Introduction **5**

2 *Rust Language **6**

2.1 Source File . . . 7
2.2 Type System . . . 7
2.2.1 Types . . . 7
2.2.2 Casting . . . 8
2.2.3 Coercions . . . 9
2.2.4 Type Coalescing . . . 9
2.3 Functions . . . 10
2.4 Expressions . . . 11
2.4.1 Literals . . . 11
2.4.2 Operators . . . 12
2.4.3 Block Expressions . . . 13
2.4.4 While Loops . . . 13
2.4.5 If Expressions . . . 14
2.4.6 Return Expressions . . . 14
2.4.7 Path Expressions . . . 14
2.5 Enums . . . 14
2.6 Pattern Matching . . . 15
2.6.1 Patterns . . . 15
2.6.2 Let Bindings . . . 16
2.6.3 Match Expressions . . . 17
2.7 Generics . . . 17
2.8 Unsafe *Rust . . . 18
2.9 Built-in Functions . . . 18
2.10 Extern Functions . . . 19

3 LLLVM Language **19**

3.1 Source File . . . 20
3.2 Types . . . 20
3.3 Values . . . 20
3.4 Global String Literals . . . 21
3.5 Functions . . . 22
3.5.1 Control Flow Graph . . . 22
3.5.2 Static Single Assignment Form . . . 23
3.6 Instructions . . . 25
3.6.1 Terminators . . . 25
3.6.2 Integer Operations . . . 25
3.6.3 Type Conversions . . . 26
3.6.4 Memory Operations . . . 26
3.6.5 Function Calls . . . 26

3.7 Built-in Functions . . . 27

4 Compiler **27**
4.1 Architecture . . . 28
4.2 Lexical Analysis . . . 28
4.3 Syntax Analysis . . . 29
4.4 Types . . . 29
4.4.1 Type Lowering . . . 29
4.4.2 Type Checking and Type Coalescing . . . 30
4.5 SSA . . . 30
4.6 Functions . . . 31
4.7 Symbol Tables and Scopes . . . 32
4.8 Expressions . . . 33
4.8.1 Literals . . . 33
4.8.2 Operator Expressions . . . 34
4.8.3 Casting . . . 35
4.8.4 Block Expressions . . . 36
4.8.5 While Loops . . . 37
4.8.6 If Expressions . . . 37
4.8.7 Return Expressions . . . 38
4.8.8 Function Call Expressions . . . 39
4.9 Enums . . . 39
4.9.1 Memory Representation and Construction . . . 39
4.9.2 Copying Enums . . . 40
4.9.3 Returning Enums from Functions . . . 40
4.10 Destructuring and Binding . . . 40
4.11 Match Expressions . . . 42
4.11.1 Control Flow and Pattern Matching . . . 42
4.11.2 Exhaustiveness Checking . . . 45
4.12 Generics . . . 47
4.12.1 Generic Functions . . . 47
4.12.2 Generic Enums . . . 48
4.13 Unsafe . . . 49
4.14 Built-in Functions . . . 49
4.15 Extern Functions . . . 49

5 Emulator **50**
5.1 Architecture . . . 50
5.2 Lexical Analysis . . . 51
5.3 Syntax Analysis . . . 51
5.4 Memory . . . 52
5.4.1 Memory Model . . . 52
5.4.2 Memory Allocator . . . 53
5.5 Interpreter . . . 54

5.6 Built-in Functions 55

6 Library **56**
6.1 Common Data Structures 56
6.2 Low-Level Memory Operations 57
6.3 Input and Output 58

7 Performance **58**

8 Conclusion **60**

1 Introduction

Self-reference is a recurring concept in computer science, arising whenever a system can refer to, represent or operate on itself. It appears in areas ranging from computability theory to programming languages. A practical example of this concept is the Selfie project[1], which combines a self-compiling C compiler, a RISC-V emulator and a self-hosting hypervisor. Selfie uses two language subsets: C*, a subset of C, and RISC-U, a subset of the RISC-V instruction set architecture. Since the system is implemented entirely in C*, it can compile its own source code into RISC-U and execute the resulting program through emulation. The emulated program is itself capable of compiling the same source code, allowing the cycle to be repeated indefinitely, at least in theory [2]. This thesis draws inspiration from Selfie and pursues a similar goal using different languages.

The languages chosen for this work are Rust and LLVM Intermediate Representation (LLVM IR). Rust is a systems programming language that provides strong type safety and memory-safety guarantees, making it suitable for implementing systems software such as compilers and emulators. LLVM IR, meanwhile, is an intermediate representation used by compiler toolchains such as `clang` [4] and `rustc` [6]. The LLVM toolchain supports a wide range of target architectures and provides facilities for optimising LLVM IR and generating native machine code.

The central research question of this thesis is which subsets of Rust and LLVM IR are sufficient to implement a system capable of compiling and emulating itself. To investigate this question, this thesis defines suitable subsets of both languages, implements a compiler and emulator, then demonstrates that the resulting system can compile and emulate itself.

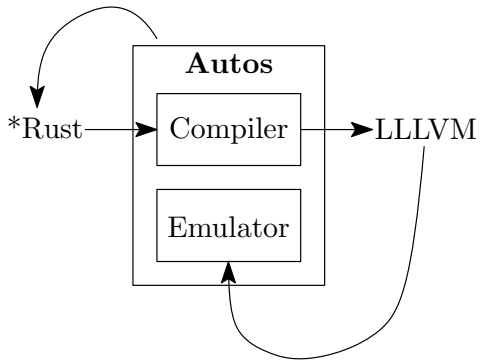


Figure 1: Autos compiles *Rust programs, including itself, and the generated LLLVM can then be emulated by it.

The resulting system, called Autos, makes use of two languages: *Rust and LLLVM, subsets of Rust and LLVM IR respectively. Autos prioritises correctness and simplicity

[1] `https://github.com/cksystemsteaching/selfie`

over runtime performance, as its primary objective is to demonstrate the feasibility of self-compilation and self-emulation. As illustrated in figure 1, Autos can compile *Rust programs, including its own source code, into LLLVM and execute the resulting programs.

The complete source code of Autos is available on a GitHub repository[2]. Bootstrapping requires an existing Rust compiler, typically `rustc` [6]. Assuming the source code is located at `src/main.rs`, Autos can be compiled as follows:

```
rustc src/main.rs -o autos -O
```

Once bootstrapped, Autos can compile and emulate itself repeatedly. For example, the following command compiles its source code, emulates the resulting LLLVM to compile itself again and writes the final output to a file:

```
./autos -c src/main.rs -e 100 -c src/main.rs -o autos.ll
```

This process can be repeated arbitrarily many times, limited only by the available time and memory. Each iteration emulates the generated LLLVM to execute another compilation cycle, demonstrating the self-referential behaviour at the heart of Autos.

2 *Rust Language

The design of *Rust, the subset of Rust developed in this work, aims to balance implementation complexity with the ability to express programs in a style characteristic of Rust. An initial design considered reducing the language to the smallest possible subset that would make implementing the self-referential system feasible. Although this would simplify the compiler, it would also make many operations dependent on unsafe code, resulting in a language that much more closely resembles C than Rust. Since Rust's primary design goal is to minimise the amount of unsafe code, this was rejected.

Another approach considered, is restricting *Rust to the safe subset of Rust and relying on (a subset of) the Rust standard library for safe abstractions. However, this approach was also rejected because supporting the standard library would increase the complexity significantly.

The resulting design therefore occupies a middle ground: *Rust includes unsafe operations, but also provides the language features necessary to build safe abstractions around them. In Rust, such abstractions are commonly implemented by encapsulating unsafe operations within data structures and exposing safe interfaces to their users.

[2] `https://github.com/scrawfordlk/autos/`

The design of *Rust is consequently guided by two competing goals: minimising the implementation effort while retaining the programming style and abstraction mechanisms that characterise Rust. This trade-off is particularly relevant because the compiler itself is implemented in *Rust. Adding a language feature can make the compiler easier to implement, but also increases the complexity of the compiler that must support that feature.

A context-free grammar, represented using EBNF, describing the complete syntax of *Rust is available in the Autos repository[3]. The grammar snippets included throughout this section are simplified versions of this grammar, adapted for readability. Unless stated otherwise, *Rust follows the semantics of Rust.

2.1 Source File

A *Rust source file consists of ASCII characters. This differs from Rust, which interprets source files as UTF-8-encoded Unicode text [7].

2.2 Type System

*Rust retains the nominal, static type system of Rust [7], though it supports a substantially smaller set of types. Every expression has a type, which dictates the set of possible values and their interpretation.

2.2.1 Types

type → "`usize`" | "`bool`" | pointer | "`u8`" | "`char`" | "`&str`" | "`()`" | "`!`" | enumtype

pointer → ("`&`" ["`mut`"] | "`*`" "`mut`") type

enumtype → identifier ["`<`" type "`>`"]

Let $\mathcal{T}$ denote the (infinite) set of *Rust types. Each type $\tau \in \mathcal{T}$ has a corresponding domain D_τ, which is the set of all values of type τ.

The core types of *Rust are:

- `usize`: In Rust this denotes an unsigned machine-word-sized integer. In *Rust however, this is a 64-bit unsigned integer with $D_{\texttt{usize}} = \{0, 1, \ldots, 2^{64} - 1\}$, which corresponds to Rust's semantics on a 64-bit target machine.
- `bool`: A boolean with $D_{\texttt{bool}} = \{\texttt{true}, \texttt{false}\}$.

[3] `https://github.com/scrawfordlk/autos/blob/master/doc/rust/grammar.md`

- For any type $\tau \in \mathcal{T}$, `*mut` τ is a mutable raw pointer to τ.
- For any type $\tau \in \mathcal{T}$, a reference to τ can either be immutable (`&`τ) or mutable (`&mut` τ). The semantics are the same as Rust's reference semantics, i.e. they are restricted to valid memory locations.

These types form the core of *Rust because they provide the functionality required for a minimal Rust subset. `usize` is needed as a general integer type, while `bool` is necessary for conditional expressions. Raw pointers are necessary for lower-level memory access, for which Rust's mutable raw pointer is sufficient. References are included to allow memory to be accessed through Rust's safety-checked reference semantics rather than requiring raw pointers for all memory access.

The remaining types primarily improve the usability and expressiveness of *Rust. Although they increase the complexity of the compiler, they simplify the implementation of the system, including the compiler itself. These convenience types are:

- `u8`: An unsigned 8-bit integer with $D_{\texttt{u8}} = \{0, 1, \ldots, 2^8 - 1\}$. It is included for easy byte-level manipulation of memory and convenient conversions between `usize` and `char`.
- `char`: An ASCII character, represented as an 8-bit integer. Its domain consists of the 128 ASCII characters. This is unlike Rust, where `char` is a Unicode scalar value, represented as a 32-bit integer.
- `&str`: Unlike Rust, where `&str` is a dynamically sized string slice type [7], *Rust treats `&str` as a built-in enum type used exclusively for string literals. Its underlying data can only be accessed through built-in functions.
- `()`: The unit type with $D_{()} = \{()\}$, representing the absence of a value. It is included because many expression constructs do not yield a meaningful value.
- `!`: The never type with $D_{!} = \varnothing$, denoting expressions that never evaluate to a value (*diverging* expressions) [7]. `!` forms the bottom type of *Rust's coercion hierarchy (see section 2.2.3).
- An identifier referring to a defined enum. Enums may be generic, in which case their type argument must be specified. They provide a mechanism to create abstractions over unsafe code in *Rust.

2.2.2 Casting

*Rust supports a subset of Rust's casts while mostly preserving their semantics. Table 1 lists the supported casts and their corresponding source and target types.

The only exception is the cast from `u8` to `char`, which differs from Rust's semantics. In Rust, this cast yields the Unicode scalar value corresponding to the source byte, whereas in *Rust, the most significant bit (MSB) is always set to zero. Thus, the

Source type	Target type
`bool`	`usize`
`u8`	`usize`, `char`
`char`	`u8`, `usize`
`usize`	`u8`, `*mut` τ
`&mut` τ	`&`τ, `*mut` τ
`*mut` τ	`usize`, `*mut` σ

Table 1: Supported casts in *Rust.

resulting character is always an ASCII character, as any non-ASCII value is mapped to its corresponding ASCII value by clearing the MSB.

2.2.3 Coercions

Coercions are implicit conversions between types that reduce the need for explicit casts. A coercion may occur at any *coercion site*, i.e. any location where a value of a particular target type is expected. If the source type can be coerced to the expected target type, the conversion is inserted implicitly [7].

For two *Rust types τ_a and τ_b, we write $\tau_a \rightsquigarrow \tau_b$ if a value of type τ_a can be coerced to type τ_b. We further define the coercion relation $\rightsquigarrow$ as the reflexive and transitive closure of the primitive coercions listed below. Hence, $\rightsquigarrow$ forms a preorder (even a partial order) over the set of *Rust types.

The primitive coercions of *Rust are:

τ_a	τ_b
`&mut` τ	`&`τ
`!`	τ

The first coercion permits a mutable reference to be used as an immutable reference. The second coercion allows diverging expressions to be used wherever an expression of any type is expected. Since $\forall \tau \in \mathcal{T} : \texttt{!} \rightsquigarrow \tau$, `!` is the least element of the partial order.

2.2.4 Type Coalescing

Certain language constructs, such as if expressions, require the types of multiple expressions to be combined into a single common type. *Rust performs this operation using *type coalescing*. For two *Rust types τ_a and τ_b, type coalescing is the partial function

$$\text{coalesce} : \mathcal{T} \times \mathcal{T} \rightharpoonup \mathcal{T}$$

defined by

$$\text{coalesce}(\tau_a, \tau_b) = \begin{cases} \tau_b, & \text{if } \tau_a \rightsquigarrow \tau_b, \\ \tau_a, & \text{if } \tau_b \rightsquigarrow \tau_a, \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

In other words, if the first type can be coerced to the second, the second type is chosen. Otherwise, if the second type can be coerced to the first, the first type is chosen. If neither coercion exists, the types cannot be coalesced and the program is ill-typed, hence the type checker rejects the program. Type coalescing is a simplified version of Rust's least upper bound coercion [7] and is used whenever *Rust requires multiple expressions to have a common type.

2.3 Functions

function → signature block

signature → ["unsafe"] "fn" identifier [generic] params ["->" type]

params → "(" [variable {"," variable} [","]] ")"

variable → pattern ":" type

Function definitions may only occur at the top level of the source file. They consist of a signature followed by a block expression, which forms the function body (see section 2.4.3). Let σ denote the type of this function body. The signature is comprised of the function's name, its parameters and optionally its return type τ. Omission of the return type implies $\tau =$ `()`. It must hold that $\sigma \rightsquigarrow \tau$.

Normally, the value of the function body is used as the function's return value. A return expression can instead terminate the function immediately and provide its return value explicitly (see section 2.4.6).

The entry point of a *Rust program is the function called `main`.

```
fn main() { ... }
```

Unlike Rust, *Rust does not provide access to the standard library for handling command-line arguments. Instead, command-line arguments can be accessed using the following alternative signature:

```
#![no_main]
#[unsafe(no_mangle)]
fn main(argc: usize, argv: *mut *mut u8) -> usize { ... }
```

This uses attributes, which are ignored by *Rust. The `#![no_main]` attribute prevents the bootstrapping Rust compiler from generating its usual entry point, while `#[unsafe(no_mangle)]` prevents it from mangling the function name. Name mangling transforms function names into unique names in the compiled binary, e.g. to distinguish different instantiations of a generic function. Consequently, these attributes ensure that the resulting binary refers to this function as `main`, allowing it to serve as the program entry point [7].

2.4 Expressions

*Rust is an expression-oriented language, like Rust. Most language constructs are expressions, which have a type and may produce a value.

2.4.1 Literals

literal → integer | boolean | character | string

integer → digit {digit}

boolean → "`true`" | "`false`"

character → "'" printable "'"

string → "\"" {printable} "\""

*Rust has four kinds of literals:

- Decimal integer literals of type `usize`. Unlike Rust, *Rust always assigns integer literals the type `usize`, rather than inferring their type from the context [7].
- Boolean literals of type `bool`.
- Character literals of type `char`.
- String literals of type `&str`.

2.4.2 Operators

expression → "`return`" [expression] | assignment

assignment → comparison ["`=`" assignment]

comparison → arithmetic [("`==`" | "`!=`" | "`<`" | "`>`" | "`<=`" | "`>=`") arithmetic]

arithmetic → term {("`+`" | "`-`") term}

term → cast {("`*`" | "`/`" | "`%`") cast}

cast → unary {"`as`" type}

unary → ("`*`" | ("`&`" ["`mut`"])) unary | factor

factor → literal | path | "`(`" expression "`)`" | ["`unsafe`"] block | if | while | match

*Rust provides operators for arithmetic, comparison, borrowing, dereferencing, type conversion and assignment. The expression grammar follows Rust's operator precedence and associativity. The semantics of most operators remain unchanged from Rust.

Although the production rule of an assignment allows any expression as the left-hand side, it must be either a mutable variable, a dereferenced mutable reference or a dereferenced raw pointer. Assignments in *Rust always evaluate to `()`.

The comparison operators are restricted to expressions of type `usize`, `u8`, `char` or `bool`. In particular, *Rust does not perform automatic dereferencing when comparing values, nor does it support comparisons of `&str` values.

Integer arithmetic is unsigned, since all integer types are unsigned, and always wraps around on overflow and underflow. Unlike Rust, where division by zero causes a panic, division by zero in *Rust is undefined, matching the semantics of LLVM IR.

The casting operator `as` follows the semantics as described in section 2.2.2.

The borrow operators `&` and `&mut` create references to values, but behave more simply than their Rust counterparts. When applied to a variable, they create a reference to that variable's memory location. When applied to an expression that is not a variable, they create a reference to a temporary copy stored on the stack. This allows an expression to be passed to functions that expect a reference.

The dereference operator `*` accesses the value stored at the referenced memory location. In *Rust, unlike in Rust, dereferencing a reference produces a value that is no longer associated with the original memory location. Applying `&` or `&mut` to the result of a dereference therefore creates a reference to a temporary copy of the value rather than

to the original memory location. Modifications through such a reference do not affect the original value. However, if a borrow operator immediately follows a dereference, as in `&*`, *Rust retains the reference to the original memory location. This behaviour is particularly important when converting a raw pointer into a reference, as the reference must point to the same memory location.

2.4.3 Block Expressions

block → "{" {(binding | expression) ";"} [expression] "}"

A block introduces a new lexical scope and consists of a sequence of statements, optionally followed by a tail expression. Statements are either let bindings (see section 2.6.2) or expression statements. A block evaluates to its tail expression, if present, or to `()` otherwise. If execution of a statement diverges, the entire block diverges and therefore has type `!`. The following example illustrates both cases:

```
{
  let x: usize = 40;
  sum(x, 2) // sum has return type usize
};
{
  exit(42); // exit has return type !
  0
};
```

The first block contains a let binding followed by the tail expression `sum(x, 2)`. Since `sum` returns `usize`, the block also has type `usize`. The second block contains an expression statement that calls `exit`, followed by the tail expression `0`. Since `exit` is a diverging function, the statement never completes and the entire block diverges.

Blocks do not only appear in the context of expressions, but also as part of other syntactic constructs. For instance, a function's body is a block.

2.4.4 While Loops

while → "`while`" expression block

A while loop is an expression consisting of a condition of type `bool` and a block of type τ, where $\tau \rightsquigarrow$ `()`. While loops always evaluate to `()`.

2.4.5 If Expressions

if → "`if`" expression block ["`else`" (if | block)]

An if expression consists of a condition of type `bool`, a then branch and an optional else branch. The optional else branch is either another if expression or a block. If the else branch is omitted, it implicitly has type `()`. The type of an if expression is obtained by type coalescing the types of both branches.

2.4.6 Return Expressions

expression → "`return`" [expression] | assignment

A return expression exits the current function, optionally returning the value of an expression. If the expression is omitted, `()` is returned. While the type of the returned expression must coerce to the return type of the function, the return expression itself has type `!`, since control flow does not continue after a return.

2.4.7 Path Expressions

path → identifier [args | "`::`" identifier [args]]

args → "`(`" [expression {"`,`" expression} ["`,`"]] "`)`"

identifier → (letter | "`_`") {letter | digit | "`_`"}

Like Rust, a sole identifier denotes a variable use, whereas an identifier followed by an argument list denotes a function call.

Although functions cannot be defined with `::` in their name, the construction of enum values and built-in functions use this syntax.

2.5 Enums

enum → "`enum`" identifier [generic] "`{`" variant "`,`" {variant "`,`"} "`}`"

variant → identifier ["`(`" type {"`,`" type} "`)`"]

instance → identifier "`::`" identifier [args]

*Rust uses enumeration types (enums) to define user-defined data types. An enum declaration introduces a new nominal type consisting of one or more *variants*, each containing a list of unnamed fields. A value of an enum type is an instance of exactly one variant. Values are constructed using path expression syntax, joining the enum type and variant with `::`, followed by the arguments corresponding to the variant's fields.

Each enum value stores a *discriminant*, an integer identifying its active variant. Since the active variant is generally not known at compile time, pattern matching is used to identify the variant and destructure its fields.

The memory layout of enums in *Rust is defined explicitly. The discriminant always occupies 8 bytes and the fields of the active variant are stored alongside it. Hence, the size of a variant is the sum of the sizes of its fields plus 8 bytes for the discriminant. The size of an enum is the size of its largest variant. Formally, for an enum type e and a variant v:

$$\text{sizeof}(e) \coloneqq \max\{\text{sizeof}(v) \mid v \in \text{variants}(e)\}$$
$$\text{sizeof}(v) \coloneqq 8 + \sum\{\text{sizeof}(f) \mid f \in \text{fields}(v)\}$$

This differs from Rust, which may e.g. optimise enum representations by using smaller discriminants or eliminating them entirely [7].

Supporting user-defined data structures is necessary for encapsulating unsafe implementations behind safe abstractions. During language design, structs and enums were considered. Structs are simpler to implement, requiring only direct field access via dot notation. Enums, in contrast, require a discriminant to identify the active variant and pattern matching to distinguish variants and destructure their fields, increasing both the language and compiler complexity. Despite this added complexity, enums were chosen because they are a central abstraction in Rust and provide greater expressive power than structs.

2.6 Pattern Matching

Pattern matching is the mechanism by which *Rust matches a value, called the *scrutinee*, against a *pattern*. If the match succeeds, parts of the scrutinee may be bound to identifiers. For a successful match, the scrutinee's type must match the pattern's type.

2.6.1 Patterns

pattern → literal | enumpat | "`mut`" identifier | "`_`"

enumpat → identifier ["`::`" identifier ["`(`" pattern {"`,`" pattern} ["`,`"] "`)`"]]

A pattern is one of the following:

- A character, integer or boolean literal, which matches iff the scrutinee's value is equal to the literal value. Integer literal patterns have type `usize` or `u8`, depending on the type of the scrutinee.
- An enum variant, which matches iff the scrutinee's active variant is the specified variant.
- A (mutable) identifier, which always matches and introduces a new (mutable) variable into the current lexical scope, binding the scrutinee's value to the identifier. Existing variables with the same name are shadowed.
- A wildcard (`_`), which always matches but discards the scrutinee's value.

If the scrutinee is an immutable or mutable reference, it is implicitly dereferenced exactly once before pattern matching is performed. Identifier patterns therefore bind to the referenced value through an immutable or mutable reference, respectively. Although this behaviour is simpler than Rust's, it is sufficient for the implementation and allows pattern matching on references.

A pattern is *irrefutable* iff it matches every value of its type; otherwise it is *refutable*. Literal patterns are always refutable, since every literal type in *Rust has more than one possible value. Identifier and wildcard patterns are always irrefutable. An enum variant pattern is irrefutable iff the corresponding enum has exactly one variant; otherwise it is refutable.

Destructuring is the process of decomposing an enum value into its fields using patterns. In Rust, these patterns may be refutable, whereas in *Rust they must be irrefutable. This restriction simplifies exhaustiveness checking of match expressions.

2.6.2 Let Bindings

binding → "`let`" variable "`=`" expression

variable → pattern "`:`" type

A let binding is a statement that introduces new variables by matching a pattern against an initialiser expression. Let bindings must be type-annotated and the pattern must be irrefutable, since execution cannot continue if the pattern fails to match.

Function parameters are also irrefutable patterns and hence can destructure function arguments directly.

2.6.3 Match Expressions

match → "`match`" expression "`{`" {arm} "`}`"

arm → pattern {"`|`" pattern} "`=>`" expression "`,`"

A match expression is the only mechanism in *Rust that permits the use of refutable patterns. It consists of a scrutinee and a sequence of *match arms*, each of which introduces a new lexical scope.

During evaluation, the arms are considered in order. The first arm containing a matching pattern is selected and its expression is evaluated to produce the result of the match expression. The type of a match expression is determined by coalescing the types of all arm expressions.

Match expressions also support *or-patterns*, which consist of multiple patterns separated by `|`. An or-pattern matches if any of the patterns matches and must not contain any identifiers.

Every match expression must be *exhaustive*, meaning that for every value of the scrutinee's type, at least one match arm must match. This guarantees that evaluation of a match expression always selects an arm and prevents match failures at runtime.

2.7 Generics

*Rust supports parametric polymorphism through generics. In contrast to Rust, the generic system is very restricted: every generic function or enum may have at most one type parameter, which must be named `T`. Since *Rust does not support traits, generic parameters cannot be constrained. Therefore, generic values can only be used in operations whose semantics do not depend on their concrete type.

Initially, generics were not intended to be part of *Rust, as including them increases the complexity of the language considerably. However, during the implementation of the compiler, it became apparent that encapsulating unsafe implementations behind safe abstractions without generics resulted in substantial code duplication. Thus, generics are included to avoid unnecessary duplication.

signature → ["`unsafe`"] "`fn`" identifier [generic] params ["`->`" type]

enum → "`enum`" identifier [generic] "`{`" variant "`,`" {variant "`,`"} "`}`"

generic → "`<`" "`T`" "`>`"

A generic function may declare a type parameter `T`, which may be used in the function's parameter types, return type and local type annotations.

Enum declarations may likewise declare a type parameter `T`, which may appear in the enum's variant fields.

path → identifier [args | "::" pathsuffix]

pathsuffix → identifier [args] | "<" type ">" (args | "::" identifier [args])

Generic functions and enums are instantiated by explicitly specifying their type argument using Rust's turbofish syntax (`::<T>`). This differs from Rust, where the type parameter is often inferred from the context [7]. For example, the built-in generic function `size_of<T>` can be called as `size_of::<usize>()`. Similarly, the `Some` variant of an `Option<T>` enum can be instantiated as `Option::<usize>::Some(42)`. Generic enum variant patterns, however, never require a turbofish, as the type argument is determined by the type of the scrutinee.

2.8 Unsafe *Rust

*Rust adopts Rust's distinction between *safe* and *unsafe* code. Safe code is restricted to operations whose memory safety can be statically verified by a compiler. Unsafe code permits additional operations, whose memory-safety cannot be verified at compile-time and whose correct use is therefore the programmer's responsibility.

*Rust provides two mechanisms for writing unsafe code. Functions may be marked as `unsafe`, making calls to them unsafe and indicating that the caller must uphold an additional safety contract. Additionally, a block marked as `unsafe` permits unsafe operations within an otherwise safe function. In *Rust, the following operations are unsafe:

- Dereferencing raw pointers.
- Calling unsafe functions.

Attempting to perform an unsafe operation outside of an unsafe context results in a compile-time error.

2.9 Built-in Functions

*Rust provides a small set of built-in functions, whose functionality requires compiler support. The following built-in functions are available:

- `str::as_ptr(&str) -> *mut u8`, which returns a raw pointer to the first byte of the string data. In Rust, the return type is actually `*const u8`, but this can be circumvented by casting the pointer to `*mut u8`.
- `str::len(&str) -> usize`, which returns the length of the string.

- `size_of<T>() -> usize`, which returns the size in bytes of type `T`.

2.10 Extern Functions

extern → "unsafe" "extern" "\"C\"" "{" {signature ";"} "}"

*Rust supports foreign function declarations using `unsafe extern "C"` blocks, which declare functions using the C ABI. Extern functions are declared but not defined within this block. Calls to extern functions are unsafe, since their behaviour is determined by external code and therefore cannot be verified by the compiler.

*Rust places no restrictions on which extern functions may be declared. However, the LLLVM emulator provides implementations for only a fixed set of functions and thus Autos supports only the following extern functions:

- `exit`,
- `open`,
- `read`,
- `write`,
- `malloc`,
- `free`.

3 LLLVM Language

LLLVM (pronounced "triple L-V-M") is a subset of LLVM IR developed in this work. Its design is primarily driven by minimality, aiming to minimise emulator complexity. LLLVM therefore includes only the constructs required by the compiler, while omitting LLVM IR features whose implementation in the emulator was not considered worthwhile. For instance, LLLVM does not feature LLVM array types, as arrays can be allocated and accessed without requiring array values as a language construct.

Minimality was not pursued at all costs, however, since doing so may shift complexity from the emulator to the compiler. For example, restricting the integer comparison instruction `icmp` to a single predicate, such as `ult` (unsigned less than), would marginally simplify LLLVM but require additional compiler logic to implement the other comparisons. Hence, the design of LLLVM underwent many changes in an effort to strike a balance between minimality and the overall effort required to implement the system.

Unlike LLVM IR, LLLVM is a regular language. The complete syntax is specified by an EBNF grammar available in the Autos repository[4]. The grammar snippets in this section are simplified excerpts and may differ from the complete grammar for readability. Unless stated otherwise, LLLVM follows the semantics of LLVM IR.

3.1 Source File

LLVM IR has three forms: a human-readable textual representation, an on-disk bitcode representation and an in-memory compiler representation [3]. Autos currently supports only the textual representation, due to it being directly inspectable, though support for bitcode could be considered in the future.

A LLLVM source code file consists of ASCII characters.

3.2 Types

type → integer | "`void`" | "`ptr`"

integer → "`i1`" | "`i8`" | "`i64`"

Unlike LLVM IR, where integer types may have arbitrary bit widths, LLLVM only features the integer types `i1`, `i8` and `i64`. These represent 1-bit, 8-bit and 64-bit integers, respectively, and are effectively unsigned, since LLLVM does not retain any signed operation from LLVM IR. LLLVM also retains LLVM IR's opaque pointer type `ptr` and the `void` type.

3.3 Values

value → constant | global | local

A LLLVM value is either an integer constant, a global name or a local name. Local names may also denote basic block labels and therefore do not necessarily refer to values.

constant → digit {digit}

Integer constants are LLLVM's only literal values. They can be used wherever a value of type `i1`, `i8` or `i64` is expected. The type of an integer constant is determined by its context.

[4] `https://github.com/scrawfordlk/autos/blob/master/doc/llvm/grammar.md`

global → "`@`" identifier

local → "`%`" identifier

identifier → identchar {identchar}

identchar → letter | digit | "`_`" | "`.`" | "`$`"

Identifiers consist primarily of alphanumeric characters. The characters `_`, `.`, and `$` are also permitted to simplify name mangling during code generation (see section 4.12 for details). No other characters are permitted, even though LLVM IR allows additional characters, as they are not needed.

In LLLVM, identifiers are prefixed with a symbol indicating whether they are global or local.

Global names are prefixed with `@` and identify global string literals or functions. When used as a value, a global name referring to a string literal denotes the address of its first byte. A function's global name cannot be used as a value in LLLVM, since LLLVM does not support function pointers.

Local names are prefixed with `%`. They are used both for basic block labels and for virtual registers. A label identifies a basic block and can only be used as the target of a control flow instruction. Virtual registers represent local values within a function and hold integer or pointer values.

3.4 Global String Literals

string → global "`=`" "`constant`" "`[`" constant "`x`" "`i8`" "`]`" "`c"`" {printable} "`"`"

Global string literals provide a convenient representation for string data in LLLVM. A global string literal defines a constant byte array in the data segment. This string is not implicitly null-terminated.

The type of a string literal is specified using LLVM's array type syntax. The number of elements must match the number of bytes of the string literal. This array syntax is used exclusively in global string literals; LLLVM does not otherwise support arrays as types or values.

The syntax for global string literals is inherited from LLVM IR's syntax for global values. LLLVM restricts this syntax to string literals, retaining only the functionality required for strings.

3.5 Functions

definition → "define" type global "(" [param {"," param}] ")" "{" {block} "}"

param → type local

block → label instruction {instruction}

label → identifier ":"

A function consists of a signature and a sequence of labelled basic blocks. The signature specifies the function's return type and its parameters. Parameters are virtual registers that are assigned the corresponding argument values when the function is called.

Each basic block is labelled, consists of a sequence of instructions and must end with a terminator instruction. A terminator instruction either returns from the function or transfers control to another basic block; control flow never falls through from one block to the next. The first basic block of a function is its *entry block*, where execution begins when the function is called.

In LLLVM the function `@main` serves as the entry point.

```
define i64 @main(i64 %argc, ptr %argv) { ... }
```

3.5.1 Control Flow Graph

A LLLVM function can be modelled as a *control flow graph* (CFG). A CFG is a directed graph

$$\text{CFG} = (B, E),$$

where B is a set of basic blocks and $E \subseteq B \times B$ is a set of directed edges. For two basic blocks $b_1, b_2 \in B$,

$$(b_1, b_2) \in E$$

if and only if control flow can transfer from b_1 to b_2. If a basic block has multiple outgoing edges, these edges may be labelled to indicate the conditions under which they are traversed. Typically, a basic block has either a single unlabelled outgoing edge or two outgoing edges labelled with 1 and 0, representing true and false respectively.

A basic block p is a *predecessor* of a basic block b if and only if $(p, b) \in E$. Conversely, a basic block s is a *successor* of b if and only if $(b, s) \in E$.

Figure 2 illustrates a CFG representing a conditional assignment. Assuming that $b \in \{0, 1\}$, if $b = 1$, control follows the left edge to the `left` basic block. Otherwise, control follows the right edge to the `right` basic block. Both basic blocks have a single outgoing edge leading to the `end` block.

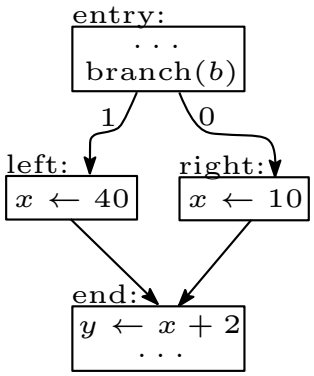


Figure 2: A CFG of a conditional assignment.

Although the branch operation branch(b) is shown explicitly in this example, it is often omitted in this thesis, with the outgoing edges alone indicating the possible control flow transitions.

3.5.2 Static Single Assignment Form

LLLVM is in *static single assignment* (SSA) form. SSA is a common representation for intermediate languages, including LLVM and GCC's intermediate representations [5]. In SSA form, every variable is assigned exactly once. For LLLVM, this means that every virtual register is assigned exactly once and cannot be reassigned within a single function.

The distinction between non-SSA and SSA form can be illustrated by the equivalent code snippets shown in figure 3. The code in figure 3a is not in SSA form because x is assigned twice. In contrast, each of x_1, x_2, and y_1 in figure 3b is assigned exactly once, so the code is in SSA form. In this example, the two definitions of x are given distinct

$$\begin{aligned} x &\leftarrow 0 \\ x &\leftarrow x + 1 \\ y &\leftarrow 2 \cdot x \end{aligned}$$

(a) Non-SSA form.

$$\begin{aligned} x_1 &\leftarrow 0 \\ x_2 &\leftarrow x_1 + 1 \\ y_1 &\leftarrow 2 \cdot x_2 \end{aligned}$$

(b) SSA form.

Figure 3: Conversion of non-SSA form (a) into SSA form (b).

names, x_1 and x_2, by numbering them. This is a common technique for converting a program into SSA form.

SSA simplifies data-flow analysis in LLVM IR because every variable has a single definition. Consequently, a virtual register always refers to the same value, regardless of

where it is used. This facilitates optimisations such as *common subexpression elimination*, which identifies equivalent subexpressions and reuses the result of one evaluation instead of computing it again. In SSA form, if two expressions perform the same operation on the same values, they produce the same result. Without SSA, this is not necessarily true because a variable may have been modified between the two expressions [5].

However, giving each definition a distinct name is not sufficient when control flow paths merge. Consider the CFG shown in figure 4. A variable x is assigned different values on two branches, causing the control flow to split. Both assignments must define distinct SSA values; otherwise, the single-definition property of SSA would be violated. In the left branch, x_1 is defined as 40, while in the right branch, x_2 is defined as 10. The two branches then merge at a common successor. A subsequent use of the value of x, such as $x + 2$, must refer to either x_1 or x_2, depending on which predecessor was taken. Conceptually, SSA represents this selection using a *ϕ-function*:

$$x_3 \leftarrow \phi(x_1, x_2).$$

A ϕ-function has one operand for each predecessor of its containing basic block and selects the value associated with the predecessor through which control entered the block.

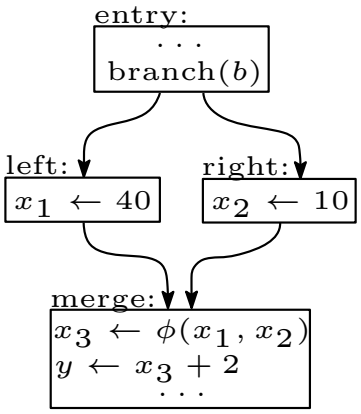


Figure 4: The ϕ-function selects the correct variable after branching control flow.

Although LLVM IR provides an explicit `phi` instruction, LLLVM does not. Instead, values that would require a ϕ-function can be represented using memory operations. This is less efficient because it requires memory accesses, but simplifies the LLLVM instruction set and its implementation. The ϕ-function is therefore only a conceptual part of the SSA model used to reason about value flow through the CFG.

3.6 Instructions

instruction → terminator | assignment | store | call

assignment → local "=" operation

operation → binary | icmp | cast | alloca | load | call

LLLVM supports a small subset of LLVM IR's instruction set, which are the operations that may be performed within a function. They can broadly be classified as terminators, assignments, memory operations and function calls. Terminators terminate the current block by either returning from the current function or jumping to another block. Assignments perform an operation and assign the result to a new virtual register. Memory operations access memory.

3.6.1 Terminators

terminator → return | branch

return → "`ret`" ("`void`" | type value)

branch → "`br`" "`label`" local | "`br`" "`i1`" value "`,`" "`label`" local "`,`" "`label`" local

These are the main control flow instructions of LLLVM. Every basic block must end with such an instruction.

`ret` terminates the current function and returns a value if the type is not `void`.

`br` transfers control to another basic block. It can be both unconditional and conditional.

3.6.2 Integer Operations

binary → ("`add`" | "`sub`" | "`mul`" | "`udiv`" | "`urem`") type value "`,`" value

icmp → "`icmp`" comparison type value "`,`" value

comparison → "`eq`" | "`ne`" | "`ugt`" | "`ult`" | "`uge`" | "`ule`"

LLLVM retains a few arithmetic instructions from LLVM IR. These operate on two values, compute a result and assign it to a new virtual register. Like in LLVM, if the result exceeds the range of the integer type, it wraps modulo 2^n, where n is the width of the integer type.

The integer comparison instruction `icmp` compares two values based on the given predicate and assigns the resulting `i1` value to a new virtual register.

3.6.3 Type Conversions

cast → (“`zext`” | “`trunc`” | “`ptrtoint`” | “`inttoptr`”) type value “`to`” type

These instructions convert a value from one type to another, assigning the result to a new virtual register.

`zext` converts an integer value to a larger integer type by zero-extending it, i.e. adding additional high bits set to zero.

`trunc` converts an integer value to a smaller integer type by discarding its most significant bits.

`ptrtoint` and `inttoptr` convert pointers to integers and integers to pointers, respectively.

3.6.4 Memory Operations

alloca → “`alloca`” type “`,`” “`i64`” constant

load → “`load`” type “`,`” “`ptr`” value

store → “`store`” type value “`,`” “`ptr`” value

`alloca` allocates memory for n elements of the specified type, produces a pointer to the beginning of the allocated memory and assigns it to a new virtual register. The elements are allocated contiguously and remain valid until the current function returns.

`load` reads a value of the specified type from the memory location addressed by the given pointer and assigns the value to a new virtual register.

`store` writes a value to the memory location addressed by the given pointer.

3.6.5 Function Calls

call → “`call`” type global “`(`” [type value {“`,`” type value}] “`)`”

The `call` instruction invokes the function identified by the given global name. If the function returns a value, the returned value can optionally be assigned to a new virtual register.

3.7 Built-in Functions

declaration → "declare" type global "(" [type {"," type}] ")"

LLLVM provides a set of functions that are treated as built-ins by the emulator. Such a function must be declared before it can be called. The following functions are supported:

- `@exit`,
- `@open`,
- `@read`,
- `@write`,
- `@malloc`,
- `@free`.

Their behaviour corresponds to that of the respective C or POSIX functions.

For efficient memory copying, LLLVM additionally supports the LLVM intrinsic `@llvm.memcpy.p0.p0.i64`. Unlike the functions described above, this intrinsic does not require an external declaration. Its signature is

```
@llvm.memcpy.p0.p0.i64(ptr, ptr, i64, i1).
```

The first argument is the destination pointer p_{dst}, the second is the source pointer p_{src} and the third specifies the number of bytes n to copy. Hence, the intrinsic copies n bytes from the memory starting at p_{src} to the memory starting at p_{dst}. The final argument is LLVM's volatile flag, which LLLVM ignores.

4 Compiler

A major part of this thesis is the implementation of a self-compiling *Rust compiler. The compiler, `starustc`, is written in *Rust and compiles *Rust programs to LLLVM. At present, only 64-bit architectures running Linux are supported. Furthermore, the current compiler does not support borrow checking, since that is outside the scope of this thesis and is also not necessary for a working compiler.

This section first presents the overall architecture of the compiler. It then briefly describes the lexical and syntax analysis phases. Finally, the lowering of specific *Rust constructs is discussed, covering both semantic analysis and code generation.

4.1 Architecture

The compiler is structured as a sequence of distinct phases. This separation of concerns makes the implementation easier to understand, while also allowing individual phases to be modified or replaced.

The compiler pipeline, visualised in figure 5, starts with lexical analysis, which scans the source program and transforms it into a stream of tokens. The next phase is syntax analysis, where the tokens are parsed and an abstract syntax tree (AST) is constructed. The AST is then passed to the semantic analysis phase, which checks whether the program is semantically valid according to the semantic rules of *Rust. Finally, the code generation phase takes the AST and lowers it to LLLVM.

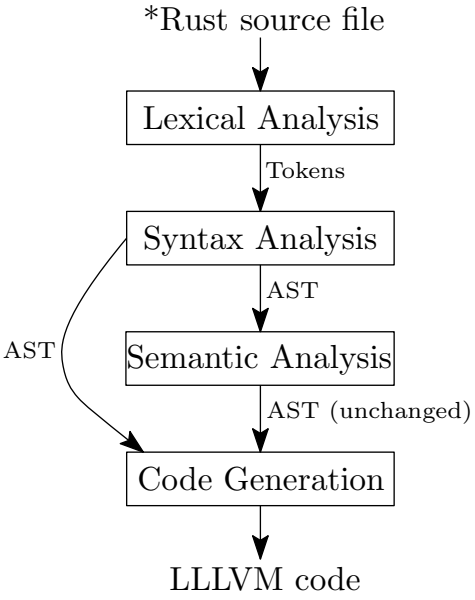


Figure 5: The compiler phases of `starustc`.

Since both semantic analysis and code generation operate on the same AST, it is possible to skip semantic analysis completely through a compiler flag. Although it is not recommended, this can lead to shorter compile times.

4.2 Lexical Analysis

In the lexical analysis phase, the lexer scans the ASCII characters of the *Rust input and groups them into tokens (also called symbols). It does this by implementing a finite state machine to scan and recognise individual tokens.

This phase is also responsible for discarding parts of the source that are not relevant to the language's syntax. This includes whitespace and line comments, but also attributes such as `#![no_main]`.

The lexer performs tokenisation lazily. Rather than scanning the entire source program in advance, the scanner scans and produces tokens on demand as requested by the parser. This allows the parser to consume the input incrementally.

4.3 Syntax Analysis

During parsing, the tokens produced by the lexical analysis phase are processed and an AST is constructed. At the same time, the parser checks whether the input conforms to the syntax of *Rust. The structure of the AST mostly reflects the structure of the grammar, retaining the information required by the following phases. For details, see the source code[5].

The grammar of *Rust is context-free and belongs to the class of LL(1) grammars. This means that it can be parsed from left to right using leftmost derivation with one look-ahead token [1]. `starustc` therefore implements a predictive parser in the form of a recursive descent parser. Recursive descent parsing is a top-down parsing technique in which the AST is constructed starting at the root and proceeding in preorder. As the grammar is LL(1), the parser requires no backtracking [1]. Each production rule is implemented as a parsing function and references to non-terminals are represented by calls to the corresponding parsing functions.

The syntax analysis phase is also responsible for skipping lifetime annotations. Since `starustc` does not currently implement borrow checking, lifetimes are not relevant to the subsequent compiler phases.

4.4 Types

4.4.1 Type Lowering

Lowering a *Rust program, which is mostly composed of expressions, involves translating the *Rust type of each expression into the corresponding LLLVM type used to represent the value of the expression. This is particularly important because LLLVM's syntax requires explicit types for most operations.

[5] `https://github.com/scrawfordlk/autos/blob/master/src/main.rs`

The mapping of built-in *Rust types to LLLVM types is relatively straightforward and can be summarised as follows:

*Rust type	LLLVM type
`usize`	`i64`
`u8`, `char`	`i8`
`bool`	`i1`
`&`τ, `&mut` τ, `*mut` τ, `&str`	`ptr`
`()`, `!`	`void`

The lowerings of `usize`, `u8` and `char` all correspond to their definitions in section 2.2.1. Although `bool` is sized one byte, it is lowered to `i1`. This is because conditional values in LLLVM are type `i1` and a 1-bit integer can fully represent a `bool` (`1` is `true`, `0` is `false`).

All pointers that represent memory addresses are lowered to `ptr`. This also applies to references, which are pointers with additional validity guarantees.

`&str` is treated by the compiler as a special built-in data structure, similar to an enum. Since enums are lowered to `ptr`, `&str` is also lowered to `ptr`. The concrete reasoning behind this is discussed in section 4.9.

Finally, the unit type `()` and the never type `!` are lowered to `void`, since neither carries any information that needs to be represented in LLLVM.

4.4.2 Type Checking and Type Coalescing

Type checking is performed by the semantic analyser and involves either checking whether two types are equal or determining whether one type can be coerced into another. The latter is implemented by exhaustively checking the *Rust coercion rules. Type coalescing relies on the same coercion relation and is therefore implemented in a similar manner.

Although type checking is handled by the semantic analyser, code generation still requires type information for various purposes, including mapping *Rust types to their corresponding LLLVM types. It therefore derives expression types independently but omits the associated checks, assuming that the semantic analyser has already verified their correctness.

4.5 SSA

A key challenge when generating LLLVM is ensuring that the resulting code adheres to SSA form, in which each SSA value is defined exactly once. Since LLLVM represents values using virtual registers, `starustc` assigns a fresh virtual register to every instruction

that produces a value. To ensure uniqueness, the code generator maintains a counter for each function, incrementing it whenever a new virtual register is allocated.

Although SSA values cannot be reassigned, *Rust supports mutable variables. `starustc` handles this by storing all variables in memory rather than keeping them exclusively in virtual registers. Each variable is allocated a memory location using `alloca`, and its initial value and subsequent assignments are written to that location using `store`. Reading a variable retrieves its current value using `load`. Thus, a variable can be mutated repeatedly without violating SSA, since each store updates the contents of a memory location rather than redefining an SSA value.

Unlike LLVM IR, LLLVM does not provide a `phi` instruction. `starustc` handles such cases using the same memory-based approach. Consider two values x and y of type τ that are produced in different branches and whose values are subsequently merged. Conceptually, this corresponds to $\phi(x, y)$. `starustc` represents this operation by allocating a memory location of type τ before the branches. Each branch then stores its respective value into this location before transferring control to the merge block, where the value is retrieved using `load`.

Figure 6 illustrates this transformation. The left side uses a `phi` instruction to select either 3 or 5, depending on which predecessor block was executed. The right side represents the same operation using memory. If the `then` branch is executed, it stores 3 into the allocated memory location; if the `else` branch is executed, it stores 5. Since every path to the merge block passes through exactly one of these branches, the subsequent `load` retrieves the value corresponding to the executed branch.

This approach can introduce runtime overhead because values that could otherwise remain in virtual registers are instead stored in memory. Since efficiency is not a primary goal of `starustc`, this trade-off is acceptable.

Throughout the remainder of this thesis, ϕ-functions are used as a conceptual shorthand for this memory-based representation, even though they are not part of the LLLVM instruction set.

4.6 Functions

Lowering a function translates a *Rust function into an equivalent LLLVM function. A function's signature is lowered by converting its return type and parameter types to their corresponding LLLVM types.

Before lowering the function body, `starustc` creates the initial basic block of the LLLVM function by emitting the label `entry`. This is necessary because LLLVM instructions may only occur within basic blocks. The function body is then lowered by compiling the block expression, as described in section 4.8.4. Compiling a block expression produces its type and, if the block yields a value, the corresponding LLLVM value.

```
define i64 @example(i1 %b) {
entry:
  br i1 %b,label %then,label %else
then:
  %x = add i64 3, 0
  br label %merge
else:
  %y = add i64 5, 0
  br label %merge
merge:
  %z = phi i64 [%x,%then],[%y,%else]
  ret i64 %z
}
```

(a) Selecting either 3 or 5 using a `phi` instruction.

```
define i64 @example(i1 %b) {
entry:
  %x = alloca i64, i64 1
  br i1 %b,label %then,label %else
then:
  store i64 3, ptr %x
  br label %merge
else:
  store i64 5, ptr %x
  br label %merge
merge:
  %z = load i64, ptr %x
  ret i64 %z
}
```

(b) Selecting either 3 or 5 using memory instructions.

Figure 6: Representing a ϕ-function using memory operations.

Every LLLVM basic block must end with a terminator. Consequently, after lowering the function body, `starustc` must ensure that the final basic block is terminated. Since this block represents the end of the function, the `ret` instruction is appropriate. The concrete return instruction emitted depends on the function's return type τ and the block type σ:

- If $\tau \in \{(), !\}$, semantic analysis guarantees that $\sigma \rightsquigarrow \tau$ and therefore $\sigma \in \{(), !\}$. The block does not produce a value, so `starustc` emits `ret void`.
- If $\tau \notin \{(), !\}$ and $\sigma \neq !$, the block produces a value. `starustc` therefore emits a `ret` containing the value produced by the block. This corresponds to the *Rust semantics in which the value of the function body is the function's return value.
- If $\tau \notin \{(), !\}$ and $\sigma = !$, the block diverges and therefore never produces a value or reaches the end of the function. `starustc` therefore emits a synthetic `ret` with an arbitrary value. This instruction is unreachable and thus has no effect at runtime.

4.7 Symbol Tables and Scopes

`starustc` uses two kinds of symbol tables to track declarations: a global symbol table for global items, namely functions and enums, and a stack of local symbol tables for local bindings. The global symbol table is maintained throughout compilation, whereas the local symbol table stack reflects the current lexical scope.

When entering a new scope, the compiler pushes an empty local symbol table onto the stack. New local bindings are inserted into the table at the top of the stack, while name lookup proceeds from the top downwards. Consequently, bindings in inner scopes take precedence over those in enclosing scopes, allowing shadowing. When leaving a scope, the compiler pops the corresponding symbol table, making its bindings inaccessible.

When compiling a function, `starustc` first pushes a symbol table for the function scope and inserts the function parameters. Since the function body is a block, compiling it pushes another symbol table. Each nested block similarly introduces a new scope, ensuring that bindings remain accessible only within their defining block and its nested blocks.

4.8 Expressions

Compiling a *Rust expression generally consists of recursively compiling its subexpressions. Each subexpression yields a type and, where applicable, a LLLVM value. The resulting types are then checked and the corresponding LLLVM operation is emitted using the resulting values. The value produced by this operation is the value of the expression.

4.8.1 Literals

Lowering literals is mostly straightforward: Integer literals are lowered to the numerically equivalent LLLVM constant. Boolean literals are represented by `1` for `true` and `0` for `false`. Character literals are represented by their corresponding ASCII value.

String literals require additional handling. First, the string data is emitted as a global string literal. Since all characters in *Rust occupy one byte, its size is equal to the length of the string literal. Each character is emitted, using character escapes where necessary, and the global string literal is assigned a unique name.

String literals are encountered during function code generation, as `starustc` does not perform a separate pass for string literals. However, global string literals cannot be emitted within a function. The code generator therefore stores all generated global string literals and emits them after all functions have been generated. Additionally, another counter is maintained to generate unique names for these literals.

The corresponding `&str` value is then constructed by first emitting an `alloca` that allocates 16 bytes. The memory layout of a `&str` is as follows:

Byte offset	Field
0	Pointer to string data
8	Length of string data

A `&str` therefore consists of a pointer to the first byte of the string data and its length. Both values are known when lowering the literal: the length is obtained from the literal itself, while the data pointer is obtained from the corresponding global string literal. These values are stored at their respective offsets, completing the construction of the `&str` value.

4.8.2 Operator Expressions

*Rust operator expressions generally map directly to LLLVM instructions. The operands are compiled recursively from left to right, producing their types and, where applicable, their LLLVM values. After type and mutability checking has verified that the operands are valid for the operator, the corresponding instruction is emitted using these values. The result of the instruction is the value of the operator expression.

Table 2 summarises the correspondence between *Rust operators and LLLVM instructions. Assignment (`=`) is not included in the table because it requires special handling: the right-hand side is compiled first, after which its value is assigned to the left-hand side. In most cases, this assignment is implemented using a `store` instruction. However, assignments to enum values are handled differently, as discussed in section 4.9. Borrowing is also omitted from the table and discussed separately below.

*Rust operator	LLLVM instruction
`==`	`icmp eq`
`!=`	`icmp ne`
`<`	`icmp ult`
`>`	`icmp ugt`
`<=`	`icmp ule`
`>=`	`icmp uge`
`+`	`add`
`-`	`sub`
`*`	`mul`
`/`	`udiv`
`%`	`urem`
`as`	`zext`, `trunc`, `ptrtoint`, `inttoptr`, `urem`
`*` (unary)	`load`

Table 2: Lowering of *Rust operators to LLLVM instructions.

Borrowing requires additional handling because its lowering depends on the expression being borrowed. If the expression is a variable, the borrow simply yields the address of that variable. If it is not a variable, the compiler allocates temporary memory on the stack using `alloca`, stores the value of the expression in this memory using `store`

and returns its address. This allows arbitrary expressions to be passed to functions by reference without requiring an explicit temporary variable in *Rust.

However, this approach does not work for all expressions. In particular, it fails when borrowing a dereferenced raw pointer, as in the following example:

```
fn borrow_raw(ptr: &*mut u8) -> &u8 {
    let ptr: *mut u8 = *ptr;
    unsafe { &*ptr }
}
```

The function converts a `*mut u8` into a `&u8` by dereferencing the reference to obtain the raw pointer, dereferencing the pointer to obtain the `u8` value and borrowing that value. Under the general borrowing rules described above, the dereference expression `*ptr` would be treated as a non-variable expression, causing the compiler to allocate temporary stack memory for the value and return its address. The resulting reference would point to memory in the function's stack frame, which becomes invalid when the function returns, leading to undefined behaviour if the reference is subsequently used.

In this case, the desired address is already available: it is the value of `ptr` itself. Therefore, `starustc` treats `&*` as a pseudo-operator which converts a raw pointer into a reference. The LLLVM value is not modified in this case, since the address does not change. This preserves the intended semantics without introducing a temporary stack allocation.

4.8.3 Casting

The lowering of a cast depends on its source and target types. `starustc` determines the required LLLVM operation directly from the *Rust types involved. If the source and target types have the same LLLVM representation, no operation is required. Integer casts are lowered to a `zext` instruction if the target type is wider than the source type and to a `trunc` instruction if it is narrower. Integer-to-pointer and pointer-to-integer casts are lowered using `inttoptr` and `ptrtoint`, respectively. For the `u8`-to-`char` cast, as outlined in section 2.2.2, *Rust clears the MSB. This is implemented using the `urem` instruction with a right-hand operand of 2^7, ensuring that the MSB is cleared. Conversely, the `char`-to-`u8` cast requires no operation, since every `char` value in *Rust is representable as a `u8`.

Table 3 summarises all casts supported by *Rust and their corresponding LLLVM operations. Since references and raw pointer are both represented by `ptr` in LLLVM, casts between them require no operation.

Any cast not covered by these cases is invalid, resulting in the semantic analyser rejecting the program.

Source type	Target type	LLLVM instruction
`bool`, `u8`, `char`	`usize`	`zext`
`usize`	`u8`	`trunc`
`u8`	`char`	`urem`
`char`	`u8`	no-op
`usize`	`*mut` τ	`inttoptr`
`&mut` τ	`&`τ, `*mut` τ	no-op
`*mut` τ	`usize`	`ptrtoint`
`*mut` τ	`*mut` σ	no-op

Table 3: Lowering of *Rust casts to LLLVM operations.

4.8.4 Block Expressions

Input: List of statements S, optional expression *tail*
Output: Pair of LLLVM value and type
diverges $\leftarrow$ false;
foreach *statement* $\in S$ **do**
  if *statement is let binding* **then**
    diverging $\leftarrow$ `compile-binding`(*statement*);
    diverges $\leftarrow$ *diverges* $\vee$ *diverging*;
  else
    $(_, type) \leftarrow$ `compile-expression`(*statement*);
    diverges $\leftarrow$ *diverges* $\vee$ (*type* $=$!);
if *diverges* **then**
  return $(_, !)$;
else if *there is tail expression* **then**
  $(result, type) \leftarrow$ `compile-expression`(*tail*);
  return $(result, type)$;
else
  return $(_, ())$;

Algorithm 1: Compilation of a block expression.

The compilation of a block expression is shown in algorithm 1, though it omits any kind of scope management for clarity. Statements are compiled in source order, while tracking whether the block diverges. Recall that a block has type `!` if any of its statements diverges. For a let binding, `compile-binding` performs the binding and reports whether it diverges, while for an expression statement, its type is checked to determine whether it diverges.

If any statement diverges, the block has type `!` and produces no value. Otherwise, if a tail expression is present, it is compiled and its LLLVM value and type are returned.

If there is no tail expression, the type `()` is returned, as that is the default value of a block.

4.8.5 While Loops

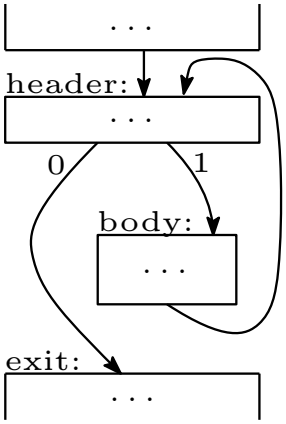


Figure 7: CFG structure of a while loop.

Code generation for while loops follows the control flow structure illustrated in figure 7, which consists of three basic blocks: a header, a body and an exit block. The header evaluates the loop condition, the body contains the loop's body expression and the exit block serves as the continuation point for code following the loop.

`starustc` first creates labels for these three blocks and emits an unconditional branch to the header. It then generates the condition in the header and emits a conditional `br` instruction that transfers control to the body if the condition evaluates to true or to the exit block otherwise.

The compiler next emits the body label and generates code for the body, which is a block expression. Once the block expression has been compiled, an unconditional `br` instruction transfers control back to the header to reevaluate the condition. Finally, `starustc` emits the exit label, allowing code generation to continue after the loop.

4.8.6 If Expressions

Code generation for if expressions is more involved than for while loops because both branches must be compiled and their resulting values merged. The control flow structure is illustrated in figure 8.

The compiler first generates code for the condition and emits a conditional `br` instruction to transfer control to either the then or else branch. The then branch is a block expression, which is compiled before an unconditional branch to the exit block is emitted. The else branch can be either a block expression or another if expression. In the latter case,

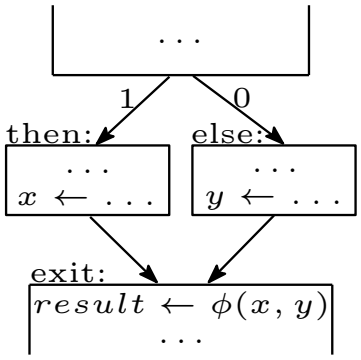


Figure 8: CFG structure of an if expression.

code generation proceeds recursively. Once both branches have been compiled, their types are coalesced to determine the type of the if expression.

The exit block merges the values produced by the two branches using a ϕ-function, yielding the value of the if expression. As described in section 4.5, this is implemented using memory rather than an explicit `phi` instruction. The compiler allocates memory to hold the result and each branch stores its value in this location before transferring control to the exit block. The exit block then loads the value, allowing execution to continue with the result of the if expression. The exception is when the if expression has type `()` or `!`, in which case no memory operations are emitted.

Since code generation is performed in a single pass, the type of the if expression is not known until both branches have been compiled and their types coalesced. Consequently, the size and type of the result's memory allocation cannot be determined when the allocation instruction is first emitted. The compiler addresses this using fixup: it initially emits an `alloca` instruction with a placeholder type, then updates the instruction to use the correct type once the type of the if expression has been determined.

4.8.7 Return Expressions

Return expressions in *Rust are lowered to LLLVM `ret` instructions. Since return expressions themselves do not yield a value, compiling one produces only the type `!`. If a return expression contains a value, the compiler generates code for that expression and checks that its type is compatible with the current function's return type. It then emits a `ret` instruction that returns the resulting value. A return expression without a value returns `()`, which normally corresponds to `ret void` in LLLVM.

The main function requires special handling. In *Rust, a main function with return type `()` terminates with exit code 0 when execution completes normally. However, the generated program's entry point must return an integer to communicate its exit status. Therefore, the compiler represents main with an integer return type when its source-level return type is `()`. Consequently, when generating a return expression in main, the

compiler must account for this difference in return types. If the return expression returns `()`, the compiler emits a `ret` instruction that returns the integer 0 instead of emitting `ret void`. This ensures that the generated program terminates with the expected exit code.

4.8.8 Function Call Expressions

Function call expressions are compiled to `call` instructions. The compiler first consults the global symbol table to verify that the called function exists and that its signature matches the provided arguments. It then compiles the argument expressions from left to right and passes their resulting LLLVM values to the `call` instruction. If the function returns a value, the result is stored in a virtual register, which represents the LLLVM value of the function call expression.

4.9 Enums

Enums require special handling in `starustc` because, unlike ordinary LLLVM values, they are represented by pointers to memory containing their discriminant and fields. This representation affects their allocation, construction, copying and function returns.

4.9.1 Memory Representation and Construction

Enums are allocated on the stack, with sufficient space for their largest variant. The compiler determines the size of this space by looking up the enum definition in the global symbol table and recursively calculating the sizes of its variants and fields, as defined in section 2.5. Memoisation avoids repeating these calculations.

Memory is allocated in units of `i64`, with the allocation size rounded up to an eight-byte boundary. To construct a variant of an enum, the required memory is allocated, the variant's discriminant is stored at offset zero and then each field is stored at its corresponding offset. Discriminant values are assigned consecutively, starting at zero for the first variant of the enum and increasing by one for each subsequent variant.

Table 4 illustrates the memory layout of an example enum, `Shape`. Its variants, `Point`, `Circle` and `Rectangle`, share the same starting address, with the discriminant at offset zero and their fields at subsequent offsets.

Field access uses byte offsets to calculate pointers to individual fields. Since LLLVM does not support pointer arithmetic directly, the compiler converts pointers to `i64` using `ptrtoint`, adds the required offset and converts the result back to `ptr` using `inttoptr`.

Byte offset	Point	Circle	Rectangle
0	Discriminant (0)	Discriminant (1)	Discriminant (2)
8		`usize`	`usize`
16			`usize`

Table 4: Memory layout of `Shape`.

4.9.2 Copying Enums

Since enums are represented by pointers but have pass-by-value semantics, moving an enum requires copying its contents rather than merely copying its pointer. The compiler emits memory-copy operations whenever such a copy is needed, including variable binding, assignment, function argument passing and construction of an enum with an enum field. This memory-copy operation was initially emitted as a helper function in each LLLVM output, but was later replaced with an intrinsic to improve performance.

The initial implementation represented enums as LLVM arrays of `i64`, using `load` and `store` to copy enums. However, arrays complicated the LLLVM emulator and introduced unnecessary overhead when copying enum values. The current implementation instead uses `alloca` to allocate multiple `i64` elements and memory-copy operations to copy enum contents, as described above. This removed the need for array values in LLLVM, which simplified the emulator immensely.

4.9.3 Returning Enums from Functions

Since LLLVM only supports returning integers and pointers, enums cannot be returned as values. Instead, `starustc` uses an indirect return mechanism. The function's return type is changed to `void` and a hidden pointer parameter `%sret` specifies where the return value must be stored. The caller must therefore allocate sufficient memory and pass a pointer to it to the callee as the first argument. The callee then constructs the enum in this memory and terminates with `ret void`. This mechanism follows LLVM's structure-return convention (*sret*) [3].

4.10 Destructuring and Binding

In `starustc`, let bindings, function parameters and match expressions share a common destructuring algorithm. For a let binding, the initialiser is compiled first and its resulting value is destructured according to the binding pattern. Function parameters are handled similarly, using the corresponding LLLVM function parameter as the value to destructure. In a match expression, the algorithm is applied to the pattern of the selected arm and the scrutinee.

The algorithm distinguishes between identifier, literal, wildcard and enum variant patterns. For an identifier pattern, memory is allocated for the bound variable, the value is stored and an entry is added to the local symbol table. This entry records the variable's *Rust type, mutability and the LLLVM value representing its memory address.

Literal patterns are ignored because they cannot be destructured, while wildcard patterns require no action. For an enum variant pattern, the global symbol table is queried to retrieve the variant's fields. The algorithm then processes each field alongside its corresponding nested pattern. Identifier patterns bind the field as described above, nested enum variant patterns are handled recursively and wildcard patterns are ignored.

Algorithm 2 formalises this procedure. In the function `destructure-enum`, *pointer* denotes a pointer to the enum field currently being processed. It is initially advanced past the discriminant and subsequently advanced by the size of each field.

Function `destructure`(*pattern, scrutinee*)`:`
- **if** *pattern is identifier* **then**
  - allocate memory, store *scrutinee* and create variable entry;
- **else if** *pattern is enum variant* **then**
  - *variant* ← `get-variant`(*pattern*);
  - *patterns* ← `get-field-patterns`(*pattern*);
  - `destructure-enum`(*variant*, *patterns*, *scrutinee*);

Function `destructure-enum`(*variant, patterns, pointer*)`:`
- $fields \leftarrow$ `get-fields`(*variant*);
- *pointer* ← *pointer* + 8;
- **foreach** $(pattern, field) \in (patterns, fields)$ **do**
  - **if** *pattern is identifier* **then**
    - allocate memory, load from *pointer*, store value and create variable entry;
  - **else if** *pattern is enum variant* **then**
    - $v \leftarrow$ `get-variant`(*pattern*);
    - *ps* ← `get-field-patterns`(*pattern*);
    - `destructure-enum`(v, *ps*, *pointer*);
  - **else if** *pattern is literal* **then**
    - throw error: nested refutable pattern;
  - *pointer* ← *pointer* + `sizeof`($field$);

Algorithm 2: Destructuring an expression using a pattern.

This algorithm omits the handling of reference scrutinees. As described in section 2.6.1, a reference is dereferenced once before pattern matching. In `starustc`, this is implemented by inspecting the scrutinee's type and emitting a `load` if it is a reference. If the scrutinee is a reference to an enum, the resulting value and the scrutinee's type information

are then passed to `destructure-enum`. Otherwise, if the pattern is an identifier, the scrutinee is bound directly to the variable.

For identifier patterns nested within enum variants, the type information determines whether the field is bound by value or by reference. When the scrutinee is a mutable or immutable reference, the field's address is stored directly as the variable's value, preserving the reference's mutability. Otherwise, the field is loaded from memory before being stored. Recursive calls propagate the type information to ensure consistent binding at deeper nesting levels.

Memory allocation also requires special handling. Emitting an `alloca` instruction at the binding site could allocate memory repeatedly when the binding occurs inside a loop, although only one allocation is required. Therefore, `starustc` defers these instructions by reserving their virtual register names during code generation. Once the function has been compiled, the deferred `alloca` instructions are inserted at its beginning. This ensures that each binding's memory is allocated once and reused across loop iterations.

4.11 Match Expressions

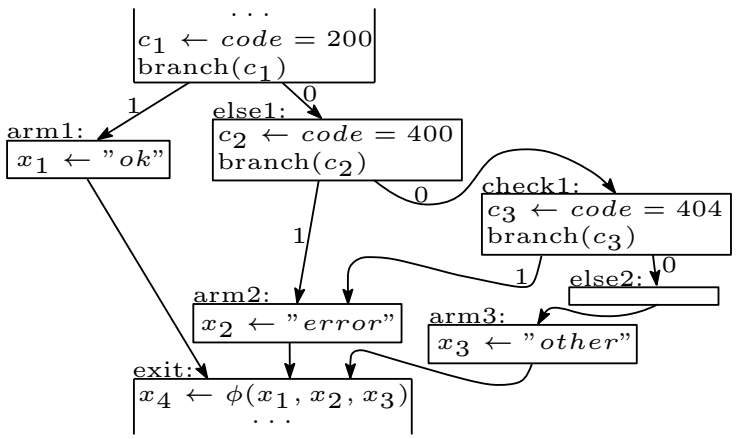
The implementation of match expressions combines control flow generation, pattern matching, destructuring, lexical scope management and exhaustiveness checking, making match expressions potentially the most complex language construct implemented by `starustc`.

4.11.1 Control Flow and Pattern Matching

The control flow of match expressions is best illustrated by an example. Figure 9 shows a match expression in subfigure 9a and its corresponding CFG in subfigure 9b.

```
match code {
    200 => "ok",
    400 | 404 => "error",
    _ => "other",
}
```

(a) Example match expression.



(b) Corresponding CFG.

Figure 9: CFG of an example match expression.

The CFG contains a basic block for each arm, labelled `arm1`, `arm2` and `arm3`, respectively. Execution begins by comparing the scrutinee against the first pattern, `200`. If the comparison succeeds, control transfers to `arm1`, which evaluates the corresponding expression before jumping to the `exit` block. Otherwise, control transfers to `else1`, where the first pattern of the second arm, `400`, is checked. If it matches, execution proceeds to `arm2`. Otherwise, control transfers to `check1`, which checks the second pattern, `404`. A successful comparison also transfers control to `arm2`, since both patterns belong to the same arm. If neither pattern matches, execution continues at `else2`, where the final arm is considered. This arm contains a wildcard pattern and therefore matches any remaining value. Since semantic analysis guarantees exhaustiveness, the compiler can transfer control to `arm3` unconditionally, without generating an additional (in this case redundant) pattern check. After evaluating its expression, control likewise transfers to `exit`.

The `exit` block merges the results produced by the individual arms. Although this corresponds conceptually to a ϕ-function, `starustc` implements the merge using the same memory-based technique used for if expressions.

Algorithm 3 formalises the code generation procedure. It abstracts away implementation details such as lexical scope management, special handling of enums and reference scrutinees, and result allocation fixup. Instead, it focuses on the control flow and pattern matching of match expressions.

The function `compile-match` first compiles the scrutinee, yielding a LLLVM value s and its type τ_s. It then allocates memory for the result of the match expression and generates an `exit` label for the block in which the result is retrieved. The result type τ is initially set to the bottom type `!` and subsequently updated by coalescing it with the type of each arm expression.

Each arm is compiled by `compile-arm`. The function generates a `success` label for the block containing the arm expression and an `else` label for the block in which the next arm is considered. If the current arm is the last arm, exhaustiveness guarantees that it must match, so the compiler emits an unconditional jump to its `success` block.

For every other arm, the compiler generates a sequence of pattern checks. For each pattern p, it determines the failure destination: if p is not the final pattern of the arm, a new `check` label is generated for the next pattern in the same arm; otherwise, failure transfers control to the `else` block.

The pattern check depends on the pattern's kind. For a literal pattern, the compiler compares the literal's value with the scrutinee using an integer equality comparison and branches to either the `success` block or the failure destination. For an enum variant pattern, it loads the scrutinee's discriminant and compares it with the discriminant associated with the pattern's variant. Identifier and wildcard patterns always match, so they require only an unconditional jump to the `success` block. If another pattern follows, the compiler emits the `check` label before proceeding to compile the next pattern.

Function `compile-match`(*scrutinee, arms*)**:**

- $(s, \tau_s) \leftarrow$ `compile-expression`(*scrutinee*);
- $ptr \leftarrow$ `emit-alloca`(...);
- $exit \leftarrow$ `new-label`("exit");
- $\tau \leftarrow$!;
- **foreach** $arm \in arms$ **do**
  - $patterns \leftarrow$ `get-patterns`(*arm*);
  - $expression \leftarrow$ `get-arm-expression`(*arm*);
  - $\sigma \leftarrow$ `compile-arm`(*patterns*, *expression*, s, *ptr*, *exit*);
  - $\tau \leftarrow$ `coalesce`(τ, σ);
- `emit-label`(*exit*);
- $result \leftarrow$ `emit-load`(*ptr*, τ);
- **return** $(result, \tau)$;

Function `compile-arm`(*patterns, expression, s, ptr, exit*)**:**

- $(success, else) \leftarrow$ (`new-label`("arm"), `new-label`("else"));
- **if** *is last arm* **then**
  - `emit-jump`(*success*);
- **else**
  - **foreach** $p \in patterns$ **do**
    - $fail \leftarrow \begin{cases} \texttt{new-label}(\text{"check"}), & \text{if } p \text{ is not the last pattern,} \\ else, & \text{otherwise.} \end{cases}$
    - **if** p *is literal* **then**
      - $value \leftarrow$ `value`(p);
      - $condition \leftarrow$ `emit-icmp-eq`($s, value$);
      - `emit-branch`(*condition*, *success*, $fail$);
    - **else if** p *is enum* **then**
      - $discriminant \leftarrow$ `emit-load`(s, `usize`);
      - $value \leftarrow$ `get-variant-discriminant`(p);
      - $condition \leftarrow$ `emit-icmp-eq`(*discriminant*, *value*);
      - `emit-branch`(*condition*, *success*, $fail$);
    - **else**
      - `emit-jump`(*success*);
    - **if** p *is not last pattern* **then**
      - `emit-label`($fail$);
  - `emit-label`(*success*);
  - `destructure`($patterns[0], s$);
  - $(result, \sigma) \leftarrow$ `compile-expression`(*expression*);
  - `emit-store`($result, ptr, \sigma$);
  - `emit-jump`(*exit*);
  - **if** *is not last arm* **then**
    - `emit-label`(*else*);
- **return** σ;

Algorithm 3: Code generation of a match expression.

Once the pattern checks have been generated, the compiler emits the `success` label and performs any required destructuring, as described in section 4.10. It uses the arm's first pattern for this purpose because semantic analysis guarantees that an arm containing a binding pattern has only one pattern. The compiler can therefore use this pattern directly to introduce the arm's bindings.

Next, the arm expression is compiled, yielding a LLLVM value `result` and a type σ. The result is stored in the allocated memory, which serves as the destination for the values produced by the different arms. An unconditional branch then transfers control to `exit`. If another arm follows, the compiler emits the `else` label before proceeding to compile that arm.

After all arms have been compiled, `compile-match` emits the `exit` label, loads the result from memory using the final coalesced type τ and returns the resulting LLLVM value and type.

4.11.2 Exhaustiveness Checking

As mentioned above, semantic analysis verifies that every match expression is exhaustive. This guarantee simplifies code generation and ensures that a match expression cannot fail at runtime, though `starustc` has yet to fully implement it. The following exhaustiveness algorithm assumes that nested patterns are irrefutable, as they are in *Rust.

More formally, given a list of patterns P and a scrutinee type τ, the algorithm determines whether there exists a value $x \in D_\tau$ that matches none of the patterns in P, where D_τ denotes the domain of τ. It also detects patterns that are unreachable because an earlier pattern already matches the corresponding value. Algorithm 4 summarises this procedure.

Input: Patterns P, domain D_τ
$C \leftarrow [0]^{|D_\tau|}$;
foreach $p \in P$ **do**
  if *p is identifier or wildcard* **then**
    catchall $\leftarrow$ true;
    report remaining patterns unreachable;
  $i \leftarrow \text{value}(p)$;
  $C[i] \leftarrow C[i] + 1$;
foreach $c \in C$ **do**
  if *not catchall* $\wedge\ c = 0$ **then**
    report non-exhaustive;
  if $c > 1$ **then**
    report unreachable pattern;

Algorithm 4: Match expression exhaustiveness algorithm.

The algorithm first initialises an array C of zeros with length $|D_\tau|$. For most types, D_τ is the set of possible values of type τ. For enum types, it is instead the set of variants, so the array's length equals the number of variants.

The algorithm then processes the patterns in source order. If a pattern is an identifier or wildcard, it matches every value of the scrutinee type. The match expression is therefore exhaustive and any subsequent patterns are unreachable, so the variable *catchall* is set to true. Otherwise, the algorithm applies the partial function value, defined as follows:

$$\text{value}(p) = \begin{cases} \text{literal value}, & \text{if } p \text{ is a literal pattern}, \\ \text{discriminant value}, & \text{if } p \text{ is an enum variant pattern}, \\ \text{undefined}, & \text{otherwise}. \end{cases}$$

The resulting value is used as an index into C and the corresponding counter is incremented. After all patterns have been processed, the algorithm examines each counter c. If $c = 0$ and there was no identifier or wildcard, the corresponding value is not matched by any pattern, so the match expression is non-exhaustive. If $c > 1$, multiple patterns match the same value, meaning the latter patterns are unreachable and are therefore redundant.

The correctness of this procedure follows from the restricted forms of patterns supported by *Rust. Since nested patterns must be irrefutable, matching a literal or an enum variant reduces to checking a single value or variant, respectively. Each such pattern therefore covers exactly one element of the scrutinee's domain. The array C records how many patterns cover each element: a zero counter identifies an uncovered value, whereas a counter greater than one identifies a value covered by multiple patterns. Because patterns are processed in source order, the first pattern covering a value is reachable, while any subsequent pattern covering that same value is redundant. An identifier or wildcard covers the entire domain, hence the algorithm can terminate immediately when either is encountered. Thus, for *Rust match expressions, this algorithm is sufficient to determine exhaustiveness and detect unreachable patterns.

In practice, the size of C is manageable for most *Rust types. For example, a `u8` has 256 possible values, so its corresponding array contains 256 elements. However, a `usize` has 2^{64} possible values, making an array of that size impractical. To avoid this problem, a heuristic can be used for `usize`: if no identifier or wildcard pattern is present, the match expression is treated as non-exhaustive. This is a practical approximation, since a sufficiently large collection of literal patterns could theoretically cover the entire domain, though it is very unlikely.

Another limitation arises when matching on `char`. In *Rust, `char` represents ASCII characters, whereas in Rust it represents Unicode scalar values. Hence, a match expression that enumerates all ASCII characters may be considered exhaustive by `starustc` but rejected by the bootstrapping compiler, which uses Rust's larger `char` domain.

When bootstrapping, match expressions on `char` must therefore include an identifier or wildcard pattern to be accepted by the bootstrapping compiler.

4.12 Generics

Although support for generics is conceptually straightforward, implementing it required changes throughout `starustc`, as generic parameters can appear wherever a type is used. During development, many bugs, that were difficult to trace, stemmed from failing to instantiate a generic type parameter in a particular part of the compiler. The main challenge was therefore not the underlying concept but ensuring that generic types were handled consistently across all relevant parts of the compiler.

4.12.1 Generic Functions

Generic functions are implemented using *monomorphisation*, whereby a separate function is generated for each concrete type argument. To achieve this, generic type parameters are represented by placeholder types during compilation. In *Rust, `T` is reserved for this purpose and can therefore be used as a placeholder without conflicting with user-defined types.

Generic functions are initially skipped during code generation and are instead only generated when called during code generation of another function. Since generic function calls use the turbofish syntax to specify their type argument, the compiler can determine which concrete type to substitute for `T`. When a generic function is called, `starustc` first saves the current state of code generation. It then temporarily substitutes the generic type parameter with the specified type and generates the corresponding function. Once code generation for the instantiated function is complete, the compiler restores the previous state and reverts the type substitution, allowing code generation for the caller to continue. To prevent duplicate code generation, `starustc` also maintains a record of which generic functions have already been instantiated for each type argument.

This approach also ensures that generic function calls within generic functions are handled correctly. When a generic function calls another generic function, the compiler recursively generates the required instantiation. Before generating the callee, its type argument must be instantiated using the caller's type argument, as it may depend on the generic type parameter `T`. This ensures that the correct concrete type is propagated throughout the call chain, even when a generic function is reached indirectly from a non-generic function.

Name Mangling Each instantiated function requires a unique name that does not collide with either another instantiation or a non-generic function. Simply appending the type argument to the function name would not guarantee this: for example, instantiating a generic function `pass<T>` with `usize` and naming the result `pass_usize` could conflict

with a user-defined function of the same name. The compiler therefore uses a simple name mangling scheme that aims to prevent collisions whilst producing valid LLLVM identifiers.

The mangled name consists of the original function name, followed by a dot and the type argument. Since type names may contain characters that are invalid in LLLVM identifiers, these characters are replaced according to a fixed mapping, as shown in Table 5. The replacements accommodate characters arising from built-in functions, generic types, references and pointer types. Spaces are replaced with underscores, which is relevant for mutable references and raw pointers. The scheme is deliberately simple, as its primary purpose is to produce valid LLLVM identifiers whilst avoiding name collisions.

Original character	Replacement
:	.
<	.
>	.
&	$
*	.
Space	_

Table 5: Character replacements used in name mangling.

4.12.2 Generic Enums

Generic enums are handled primarily through the compiler's existing type-handling mechanisms, without requiring a separate code generation process. The global symbol table records whether an enum is generic, allowing the semantic analyser to enforce that constructions of generic enums specify their type argument using the turbofish syntax.

During compilation, `starustc` stores the generic type argument of each enum type in an optional field, which remains unused for non-generic enums. For generic enums, this field records the type specified by the turbofish syntax. If this type contains `T`, it is substituted with the concrete type when code is generated for the generic function.

An important consideration is the calculation of enum sizes. As described in section 4.9.1, size calculations are memoised to avoid repeated computation. Since a generic enum can have different sizes depending on its type argument, the compiler mangles the names of generic enum instantiations to ensure that each is cached separately, as the memoisation map uses enum names as keys.

When calculating the size, the compiler resolves the generic type argument to its concrete type if it contains `T`. This ensures that the allocated memory is sufficient for the enum

value, avoiding both unnecessary memory usage and memory errors caused by insufficient allocation.

The compiler must also resolve the generic type argument when destructuring generic enums. To determine the concrete types of an enum's fields, `starustc` examines the scrutinee's type and recursively traverses it to identify the generic type argument. Once it has been identified, the compiler can translate the generic fields using their correct concrete types. This allows destructuring to work consistently with generic enums without requiring patterns to specify their generic type argument.

4.13 Unsafe

Implementing support for `unsafe` is relatively straightforward, due to its support only affecting semantic analysis.

For each unsafe operation, i.e. dereferencing a raw pointer or calling a function marked as `unsafe`, the compiler checks whether the current context is within an unsafe block or an unsafe function. If it is, the operation is permitted. Otherwise, the compiler reports an error.

The implementation maintains a depth d as part of the state of the semantic analyser, initialised to zero. It represents the current number of nested `unsafe` contexts. Hence, upon entering an unsafe block or unsafe function, d is incremented, whereas upon leaving it, d is decremented. Determining whether the current context is unsafe is therefore equivalent to checking whether $d > 0$.

4.14 Built-in Functions

Built-in functions are implemented by emitting their hard-coded definitions and inserting their entries into the global symbol table.

The exception is the generic built-in function `size_of<T>`. It returns the number of bytes occupied by a value of type `T`. Rather than generating a separate function for each possible type, calls to `size_of<T>()` are replaced directly with the size of `T`, which is known at compile time. This is also more efficient as the function call is effectively inlined.

4.15 Extern Functions

Code generation for external functions is straightforward. All *Rust types are mapped to their corresponding LLLVM types and a function declaration (`declare`) with the resulting signature is emitted. Since *Rust places no restrictions on externally declared functions, no semantic checking is required. For each external function, a new global

symbol table entry is created, which stores the signature and marks the function as `unsafe`.

5 Emulator

In addition to the compiler `starustc`, a major component of this thesis is the LLLVM emulator `llli`. Together, the compiler and emulator allow programs written in *Rust to be compiled and emulated, including the system itself.

The emulator executes LLLVM programs using a tree-walking interpreter. Although architecturally distinct from the compiler, its implementation is closely tied to design decisions made during the development of *Rust and `starustc`, as these determine the LLLVM code it must handle. For example, using LLVM arrays to implement enums increased the emulator's complexity considerably. Once arrays were deemed unnecessary, the emulator could be simplified accordingly.

Although performance was not a primary design goal, poor performance remained undesirable. This became particularly apparent towards the end of the project, when attempting to emulate the compiler or even the emulator itself. The resulting performance challenges are discussed in section 7.

This section describes the emulator's architecture, lexical and syntax analysis, memory model and allocator. It concludes with the interpreter and the implementation of built-in functions.

5.1 Architecture

The emulator processes LLLVM programs through a sequence of phases, each responsible for a distinct part of execution. This structure separates source processing from interpretation and allows the individual phases to be developed independently.

As illustrated in figure 10, the pipeline begins with lexical analysis, which converts the source program into a stream of tokens. Syntax analysis parses these tokens into an abstract syntax tree (AST) and performs most of the semantic analysis. The interpreter then traverses the AST, starting at the function `@main`. Execution continues until `@main` returns or the program terminates by calling the `@exit` function.

Since lexical and syntax analysis are completed before interpretation begins, the source program is only processed once. The interpreter can therefore operate directly on the AST without repeating these steps. Performing as much semantic analysis as possible during parsing further reduces runtime overhead by avoiding repeated validity checks.

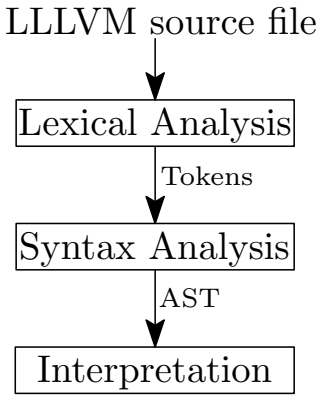


Figure 10: The phases of the emulator `llli`.

5.2 Lexical Analysis

The scanner follows the same general approach as the one implemented in `starustc`, but is adapted to LLLVM's simpler syntax. It reads the input as ASCII characters and implements a finite state machine (FSM) to identify and produce tokens.

As in the compiler, tokenisation is lazy: the scanner generates tokens only when requested by the parser, rather than processing the entire input in advance. This allows the parser to consume the source incrementally.

5.3 Syntax Analysis

The parser converts the token stream into an AST, like in `starustc`, but also performs semantic analysis, in particular type checking.

Unlike the compiler, where separation of concerns was given greater priority, the emulator benefits from performing as much semantic analysis as possible before interpretation. Since the interpreter repeatedly traverses the AST, checking semantic validity during execution would introduce unnecessary overhead, especially in frequently executed code. Performing these checks during parsing avoids repeating them at runtime and reduces the information that must be retained in the AST.

The AST's exact structure is available in the source code[6]. Its structure generally follows the LLLVM grammar, while its data structures are chosen with performance in mind. For example, functions are stored in a map rather than a list, allowing function lookups, which are required for function calls, to take expected $O(1)$ time rather than $O(n)$ time, assuming a hash map implementation.

[6] `https://github.com/scrawfordlk/autos/blob/master/src/main.rs`

The LLLVM parser is considerably simpler than the *Rust parser because LLLVM is a much simpler language. Its regular grammar allows parsing to be implemented as a FSM, whereas the *Rust parser must handle a context-free grammar and consequently requires a more sophisticated parsing approach.

5.4 Memory

The emulator manages its own memory using a simplified memory model. An alternative would have been to represent memory objects directly using data structures such as maps, without modelling memory as a contiguous address space. However, an explicit memory model was chosen, as it more closely reflects how a physical machine organises and accesses memory. This approach also provides a consistent basis for implementing memory operations and dynamic allocation within the emulator. This section describes the memory model and its initialisation, followed by the implementation of the memory allocator.

5.4.1 Memory Model

The emulator's byte-addressed memory is divided into three segments, as illustrated in figure 11. The static data segment occupies the lowest addresses and stores global values. Since LLLVM only supports global string literals, this segment contains the strings defined in the program. Above it, the heap grows towards higher addresses as memory is allocated through `@malloc`. The stack occupies the highest addresses and grows towards lower addresses.

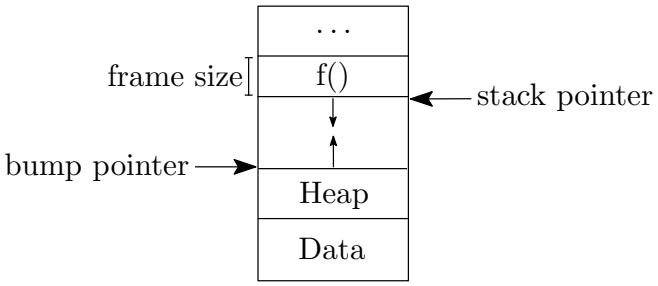


Figure 11: The emulator's memory model.

Although similar to conventional memory models, this arrangement differs in one important respect: it has no code segment. The program is instead represented by the AST, which `llli` uses directly during interpretation.

The emulator manages its memory using three state variables. The stack pointer identifies the lowest address currently occupied by the stack. Heap allocation is managed by a bump pointer, which initially points to the first free address after the static data

segment and advances towards higher addresses with each allocation. Since the global symbol table records the address of every global string literal, the boundary between the static data and heap segment does not need to be tracked separately. The third variable, the stack frame size, records the size of the current stack frame. Tracking this size allows a frame to be deallocated by advancing the stack pointer by the recorded frame size.

Initialisation The emulator's memory size is specified by the user and remains fixed during execution. Once the memory has been allocated, the emulator initialises its state variables and populates the static data segment. The stack pointer is set to the memory size, initially pointing one address beyond the valid memory range at its upper boundary, and the stack frame size is set to zero. All global string literals are then copied into memory, starting at address zero. The bump pointer is set to the next free address after these strings, marking the beginning of the heap.

Finally, the command-line arguments are placed on the stack. The argument count `argc` does not need to be stored in memory, as its value can be assigned directly to the corresponding virtual register parameter. Storing `argv`, however, requires additional preparation. First, the required space is calculated by summing the sizes of all arguments and the eight bytes needed for each pointer in the pointer array. This determines the new stack pointer. The arguments are then stored in memory, with the pointer array at the lowest addresses and the argument strings immediately above it, with each pointer referencing its corresponding string. Since the *Rust library's `String` type is not null-terminated, an additional byte must be allocated for each argument string and set to zero to terminate it.

5.4.2 Memory Allocator

Initially, the emulator did not support `@free`, so memory allocated through `@malloc` would always leak. Allocation was therefore straightforward: requesting n bytes simply advanced the bump pointer by n bytes, with the newly allocated region returned to the caller. However, the resulting accumulation of unused allocations eventually caused performance problems, as discussed in section 7. Support for `@free` was consequently introduced, requiring a more sophisticated allocation strategy.

The resulting memory allocator combines a bump pointer with a free-list, which is searched using first-fit. This approach was chosen for its simplicity.

When `@malloc` is called to allocate n bytes, the allocator reserves $n + 16$ bytes rather than just n. The additional 16 bytes store metadata associated with the allocation: the block size at offset zero and a pointer to the next block in the free-list at offset eight. The latter is initialised to zero. The pointer returned to the caller refers to the first byte after this metadata, as illustrated in figure 12.

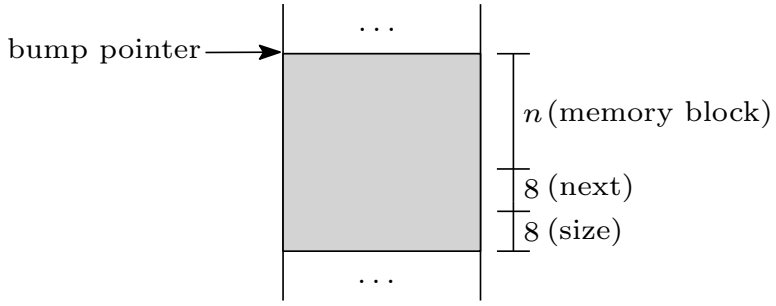


Figure 12: Memory block layout, with the block size stored at offset zero and the free-list pointer at offset eight. The allocator reserves $n + 16$ bytes for an allocation of n bytes.

When `@free` is called, the supplied pointer is used to access the block's metadata. The freed block is then added to the free-list: the address of the current head is stored in the block's next-block field and the block itself becomes the new head.

For subsequent allocations, the free-list is searched, instead of immediately advancing the bump pointer. Starting at the head, the allocator follows the pointers and checks each block's size until a suitable block is found. The first block large enough to satisfy the request is removed from the free-list and reused. If no suitable block exists, the bump pointer is advanced to allocate a new block. Best-fit allocation was also considered, but it showed no noticeable performance improvement over first-fit and was therefore not adopted.

5.5 Interpreter

The LLLVM AST is executed by a tree-walking interpreter that begins at the AST node corresponding to `@main` and traverses the tree as it executes the program. Figure 13 illustrates the interpreter's control flow.

Execution begins by locating `@main` in the AST, selecting its first basic block and executing its instructions in sequence until a control flow instruction is encountered: `br`, `call` or `ret`. A `br` instruction transfers execution to the specified basic block, whereas a `call` instruction causes the interpreter to save the current frame size, reset it to zero, look up the callee in the AST and begin executing its first basic block. Upon encountering `ret`, the interpreter returns to the calling function, optionally passing back a return value, advances the stack pointer by the current frame size, restores the previous frame size and resumes execution at the instruction following the `call`. If the return occurs in `@main`, the interpreter instead terminates and uses the returned value as the program's exit code.

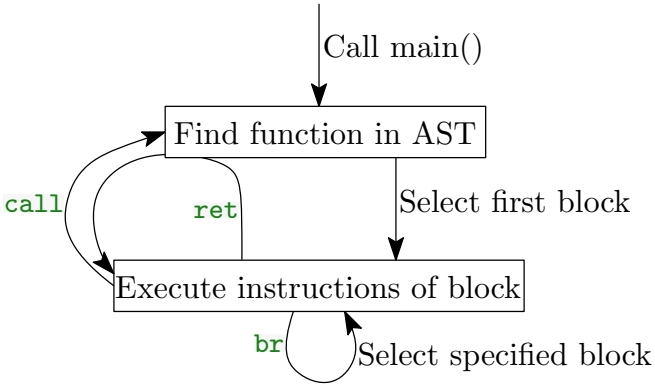


Figure 13: A flow graph of the tree-walking interpreter.

`llli` represents each value using a `usize` and a LLLVM type. This representation is sufficient to store any value supported by LLLVM. Since `usize` can represent values larger than those permitted by smaller integer types, the interpreter applies wrapping to ensure that values remain within the range allowed by their type. Wrapping is applied to every literal, to the result of each assignment and during conversion operations.

The remaining instructions are straightforward to implement in *Rust due to the intentional semantic correspondence between *Rust and LLLVM:

- Arithmetic and comparison instructions use their *Rust counterparts directly. For example, `add` is implemented using `+`, so *Rust's `+` operator is itself implemented using `+`, achieving one kind of self-reference that Autos aims for.
- Casting instructions wrap the input according to the target type and associate it with that type, effectively truncating the value when the target type is smaller and leaving it unchanged when larger.
- The `alloca` instruction allocates the specified amount of stack memory by moving the stack pointer and updating the frame size, returning the new stack pointer as the allocated memory's address.
- The `load` and `store` instructions read and write bytes at the specified address by indexing into the emulator's memory.

For each function a map of virtual registers is maintained, mapping register names to their values. Whenever an instruction assigns a value to a virtual register, the interpreter inserts that register and its value into the map.

5.6 Built-in Functions

Seven built-in functions are implemented by the emulator:

- `@exit`: A flag containing the exit code is set. The emulator periodically checks this flag and, if set, forces the current function to return. This process propagates through the call chain until the main function returns the specified exit code.
- `@open`, `@read`, `@write`: The arguments are forwarded to the corresponding *Rust extern functions and the values returned by these calls are passed back to the emulator. Since LLLVM pointers refer to addresses within the emulator's memory rather than actual memory addresses, they must first be translated. This is achieved by adding each pointer's offset to the base address of the emulator's memory, yielding a valid pointer to the corresponding memory location.
- `@malloc`: The requested amount of memory is allocated as described in section 5.4.2.
- `@free`: The memory associated with the given pointer is freed as described in section 5.4.2.
- `@llvm.memcpy.p0.p0.i64(ptr, ptr, i64, i1)`: The specified number of bytes is copied from the source to the destination using the *Rust library's `memcopy` function, which implements the same semantics. Before the call, the LLLVM pointers are translated into actual memory addresses. As mentioned in section 3.7, the last argument is ignored.

6 Library

Since the Rust standard library is unavailable in Autos, the project requires a custom library to support the implementation. A small *Rust library was therefore developed, providing a range of functionality, including common data structures, mathematical operations, character handling, low-level memory operations and I/O.

This section describes the components most relevant to the implementation, focusing on the data structures and low-level operations on which the compiler and emulator rely, as well as the available I/O functionality.

6.1 Common Data Structures

Several data structures are used throughout the codebase:

- `Vec<T>`: A growable heap-allocated buffer. It initially allocates no memory and grows in powers of two as additional capacity is required.
- `Box<T>`: An owning heap-allocated pointer. It provides heap allocation through a safe interface, avoiding the need for callers to use unsafe code.

- `Option<T>`: Represents an optional value, which is either `Some(T)` or `None`. This provides an explicit representation of absence without relying on null pointers.
- `StringMap<T>`: A hash map that associates `String` values with values of type `T`. Hash collisions are handled by storing entries with the same hash in a collision list. The map is therefore implemented as a vector of collision lists, each represented by a `Vec<StringMapEntry<T>>`.
- `StringMapStack<T>`: A stack of `StringMap<T>` instances used primarily to manage lexical scopes in *Rust. It is implemented as a vector of maps, with the current scope determined by the top of the stack, which is an index of the vector.
- `String`: A growable ASCII string backed by a `Vec<u8>`.

An earlier version of the library also included `List<T>`, a singly linked list initially used for collision chains in the hash map. However, it was removed because a vector provided the required functionality with less implementation complexity.

6.2 Low-Level Memory Operations

The library provides a small set of low-level functions to support manual memory management and pointer manipulation:

- `alloc<T>(usize) -> *mut T`: Allocates memory for the specified number of elements of type `T`. Allocation failure is handled within the function, so callers do not need to check the result for allocation errors. The allocated memory is not initialised, avoiding the overhead of explicitly zeroing it. Consequently, callers must ensure that memory is initialised before it is read, which is why the function is marked as `unsafe`.
- `ptr_add<T>(*mut T, usize) -> *mut T`: Advances a mutable pointer by a specified number of elements. Since *Rust does not support pointer arithmetic directly, this function implements it by casting the pointer to `usize`, adding the offset multiplied by the size of `T` in bytes and casting the result back to `*mut T`.
- `memcopy<T>(*mut T, *mut T, usize)`: Copies a specified number of elements of type `T` from a source address to a destination address. The operation is marked as `unsafe` because invalid pointers or insufficiently sized memory regions may lead to undefined behaviour.

These functions form the foundation for the library's heap-allocated data structures. By centralising low-level memory operations, they also reduce the need to duplicate such functionality throughout the codebase.

6.3 Input and Output

The library includes a minimal I/O interface for basic program interaction and file access. It provides functions for writing output to `stdout`, writing a `String` to a file and reading a file into a `String`. These operations are sufficient for the basic input and output requirements of the compiler and emulator without requiring a full standard library.

7 Performance

High performance was never a primary goal of this project. Nevertheless, poor performance is undesirable when it threatens to make the system unusable. This section discusses the main performance issues encountered during the development of Autos and the measures taken to address them.

Performance became a concern only when the first attempt at self-emulation was made, by which point self-compilation was already working. Self-emulation was extremely slow and eventually exhausted the available system memory, causing an out-of-memory error on the machine it was running on, which had 32 GB of RAM. Further investigation revealed that self-compilation itself consumed approximately 6–7 GB of memory. A high memory usage was initially expected, as all memory allocations were deliberately leaked to avoid implementing `@free`, thereby simplifying the emulator's memory allocator. However, this memory usage was much too high. Extensive heap profiling identified three main causes of excessive memory consumption.

First, data structures were allocated with unnecessarily large initial capacities. For example, vectors initially reserved space for ten elements regardless of their actual requirements. Reducing these initial capacities, including setting the initial capacity of vectors to zero, reduced self-compilation's memory consumption from 6–7 GB to below 100 MB.

Secondly, the interpreter allocated several data structures for each function invocation, most notably the virtual register map maintained for each function. Since function calls are frequent, these allocations accumulated rapidly and were leaked when functions returned. Following a discussion with the supervisor, support for `@free` was introduced to deallocate memory in problematic cases such as this one, where no practical alternative was available.

Thirdly, vectors continued to leak memory despite being freed after use. The cause was the resizing mechanism, which allocated a new memory block without deallocating the previous one. Thus, freeing the vector only released its most recent allocation, leaking earlier allocations. This was resolved by modifying the vector implementation to free the previous allocation whenever a vector is resized.

These changes resolved the most critical performance issues, reducing memory consumption substantially and making self-emulation feasible. Further optimisations, while not essential, yielded significant runtime improvements at a relatively low implementation cost. One such optimisation was memoising enum size calculations. Since determining an enum's size requires recursive calculations and this information is frequently needed when allocating and copying enums, caching the results reduced redundant work and improved code generation performance.

Another significant optimisation was removing LLVM arrays from the LLLVM language. This simplified the emulator by allowing all LLLVM values to be represented using `usize`. It also made enum copying more efficient by eliminating the need for individual `load` and `store` instructions. This change caused the largest runtime improvement apart from the memory related fixes, reducing execution time by more than half.

Finally, the manually emitted memory-copy function was replaced with the LLVM intrinsic `@llvm.memcpy.p0.p0.i64`. This delegates memory copying to the emulator rather than executing it instruction by instruction in the emulated program, allowing the operation to be performed more efficiently. As copying is frequent due to the widespread use of enums, this change provided a further noticeable improvement in runtime performance.

8 Conclusion

This thesis presented two new languages: *Rust, a minimal but Rust-like subset of Rust, and LLLVM, a minimal subset of LLVM IR. The design of *Rust aims to retain the essential characteristics of Rust while keeping the language small enough to implement with a relatively simple compiler. LLLVM, in turn, was designed to serve as a straightforward compilation target while retaining the features necessary to express programs written in *Rust.

Building on these languages, the thesis introduced `starustc`, a compiler that translates *Rust programs into LLLVM. Its implementation addressed several challenges, including SSA, enums, pattern matching and generics. In addition, an emulator, `llli`, was developed to execute LLLVM programs through interpretation. This required addressing challenges related to execution performance and memory allocation, while keeping the implementation sufficiently simple for the project's objectives. Together, the compiler and emulator form Autos, a self-contained system that compiles *Rust programs into LLLVM and executes them.

Since Autos itself is implemented in *Rust, it can compile its own source code into LLLVM and execute the result using its emulator. The resulting program can then compile the source code again, allowing the process to be repeated indefinitely in principle, subject to available time and memory. Autos thus demonstrates how carefully designed language subsets can enable self-compilation and self-emulation without requiring complete implementations of Rust or LLVM IR.

Although Autos achieves its initial goal, several areas remain open for improvement. These include extending platform support beyond Linux, potentially supporting 32-bit architectures, improving semantic analysis and implementing a borrow checker to bring *Rust closer to Rust's behaviour and guarantees. Such extensions could enhance the system while retaining its emphasis on simplicity.

Autos thus demonstrates that self-reference can be realised in a compact, self-contained software system. By compiling its own source code, executing the result and repeating the process, it becomes both the instrument and the subject of its own computation.

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2 edition, 2006.

[2] Christoph M. Kirsch. Selfie and the basics. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2017, page 198–213, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3133850.3133857.

[3] LLVM Project. LLVM language reference manual. Accessed: 2026-08-05. URL: https://llvm.org/docs/LangRef.html.

[4] LLVM Project. Clang: A C language family frontend for LLVM, 2026. Accessed: 2026-07-28. URL: https://clang.llvm.org/.

[5] Jeremy Singer. Introduction. In Fabrice Rastello and Florent Bouchez Tichadou, editors, *SSA-based Compiler Design*, pages 3–12. Springer, 2022. doi:10.1007/978-3-030-80515-9_2.

[6] The Rust Project Developers. *The rustc Book*. The Rust Project, 2026. Accessed: 2026-07-28. URL: https://doc.rust-lang.org/rustc/.

[7] The Rust Project Developers. The Rust Reference, 2026. Accessed: 2026-08-01. URL: https://doc.rust-lang.org/reference/.