Agda2Rust: A Study on an Alternative Backend for the Agda Compiler

Hector Peeters
Supervisors: Jesper Cockx, Lucas Escot
EEMCS, Delft University of Technology, The Netherlands

A Dissertation Submitted to EEMCS faculty Delft University of Technology
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
22-6-2022

Agda2Rust: A Study on an Alternative Backend for the Agda Compiler

HECTOR PEETERS*, Delft University of Technology, The Netherlands

Agda is a functional programming language with built-in support for dependent types. A dependent type depends on a value. This allows the developer to specify strict constraints for the types used in an application. Writing code with dependent types results in fewer type-related errors slipping through the compilation process. When executing Agda code, it is first compiled into a different programming language. This process is called code extraction. This step is applied since most of the typed code becomes unnecessary after type-checking and can therefore be removed. Currently, the Agda compiler supports only Haskell and JavaScript as target languages. However, exploring Rust as a target language could potentially lead to better performance, and it would allow access to the Rust library ecosystem. Overall, the *agda2rust* backend could be a viable alternative to what is currently available. While it is not fully feature-complete and will not outperform any existing backends yet, *agda2rust* provides a way to seamlessly integrate algorithms and data structures implemented and proven in Agda into an existing Rust codebase.

Additional Key Words and Phrases: Agda, Rust, Code Extraction, Dependent Types

1 INTRODUCTION

When writing secure and correct software, using a language with a static type system is the norm. It has the benefit of catching many errors and bugs during compile time rather than at runtime. Compared to the conventional static type system, there is an even stricter variant, the dependent type system, used in the functional language Agda [The Agda Team 2022, What is Agda?]. Having these stricter types makes it possible to assert and prove the correctness of specific properties of a program. Once a dependently typed program is proven correct, there is a lot of unnecessary code regarding the types that is irrelevant to the execution. This code must be stripped out, leaving only the code required for executing a program. The most straightforward approach for running a dependently typed program is thus to convert this execution code to a different programming language. The process of converting one language into another is called code extraction.

There currently are code extractors for the most common dependently typed programming languages, such as the code extractor for Coq [Letouzey 2008] or the current JavaScript [The Agda Team 2022, JavaScript Backend] and Haskell [The Agda Team 2022, GHC Backend] backends for Agda. While the former has significant performance drawbacks due to its interpreted nature, the GHC backend compiles using the LLVM compiler toolchain [Lattner 2002], which outputs an optimised executable. A new backend might not necessarily bring any performance improvements, but it would add support for Foreign Function Interface (FFI). Having FFI support would allow the Agda language to call Rust functions and thus use the Rust library ecosystem. Nowadays, a lot of vulnerabilities in existing software are caused by memory-related issues. Eliminating the need for manual memory management would thus prevent a wide array of bugs. As Rust is seen as a safe programming language [Jung et al. 2021], it has the benefit that it would have a better chance of catching any bugs in the generated code. This would not be the case with a more dynamic or manual memory-managed programming language. Exploring alternative target languages could yield a superior alternative to what is currently implemented.

Author's address: Hector Peeters, Delft University of Technology, The Netherlands.

The main question of this research project is: "Is Rust a viable and efficient target language for a code extractor for Agda". As the scope of this question is quite large, it breaks down into multiple sub-questions. The central aspect of this question is how one would convert Agda-specific language features like laziness or currying to Rust. The other sub-questions can be summarised as follows:

- How do we account for the different memory models of Agda and Rust?
- For which subset of the Agda features is Rust a good target language?
- Are there concepts which cannot be converted from Agda to Rust?
- How do we eliminate as much unsafe code as possible from the generated Rust code?
- What are potential extensions or optimisations to the final extractor built during this project?
- Can the code extractor be integrated with the Rust build system?

The paper will roughly follow the same structure as the questions above. The paper will start with the necessary background information on Agda and Rust, which can be found in Section 2. Next, the method will be discussed in Section 3. The paper's main body will discuss the compiler's overall design and the Agda-specific features, as seen in Section 4. Afterwards, Section 5 will be dedicated to the results, benchmarking and performance analysis compared to other Agda backends. Moreover, at the end of the paper, possible improvements and optimisations will be laid out, and a conclusion will be drawn.

2 BACKGROUND

Before delving into the details of the compiler implementation, it is necessary to understand the basic concepts of Agda and Rust and their conceptual differences. The following two sections will provide an introduction to both of these languages.

2.1 Agda

Agda is a dependently typed, functional programming language [The Agda Team 2022, What is Agda?] loosely related to other functional languages such as Coq, Epigram, and Idris. Having dependent types [Bove et al. 2009] allows the developer to be much more expressive with the types used in a program. These types can depend on expressions like numbers or booleans, which gives the programmer the power to do things such as ensuring a binary tree is always balanced or array indices will never go out of bounds, all at compile time. An example of a function that maps a *Vec* with elements of type A to type B can be seen in Figure 1. Here, the type *Vec* depends on the natural number value n, which asserts that the input and output vectors will have the same size at compile time. If the programmer had written *map f xs* instead of *con (f x) (map f xs)*, which is incorrect, the program would not have compiled as the resulting vector would have length zero regardless of the input length.

```
map : {@0 A B : Set} {@0 n : Nat} → (A → B) → Vec A n → Vec B n
map f nil = nil
map f (con x xs) = con (f x) (map f xs)
```

Fig. 1. Dependent Types in Agda

Agda is also considered a total language [Turner 2004] which adds some additional safety features to the language. It makes sure a program will always terminate and will not throw runtime errors.

In addition to the totality of Agda, it can also be used as a proof assistant, which adds the capability of proving a program's mathematical theorems or properties. As opposed to writing tests, writing proofs for algorithms leads to less incorrect behaviour as any edge cases will have to be handled in the proof while they might be left out in the test suite.

2.2 Rust

Rust is a general-purpose systems programming language [The Rust Team 2022b] that focuses on both safety and efficiency. It uses a different memory model from more common systems languages like C or C++ [Lee and Chang 2002], which use a more manual approach to memory management. Rust employs an ownership model [Weiss et al. 2019], which adds the restriction that every piece of allocated memory can only have one owner. It also uses liveliness analysis to determine all allocations and deallocations at compilation time by tracking these owners. When the owner of the block of memory goes out of scope, the allocated memory gets deallocated. It has stricter constraints regarding references, moving, and copying values to ensure the compiler correctly understands this tracked memory. Aside from the different memory model, Rust has most of the features expected from a well-established programming language. It supports generics, closures, and unlike C, a powerful macro system that does more than simple text substitution. Additionally, as Rust is a relatively young language, most of its libraries have been written from scratch with more recent technologies. This, along with its package manager Cargo, comparable to the Node Package Manager [The NodeJS Team 2022], provides the programmer with a comprehensive set of tools for building secure and performant applications.

3 METHOD

When writing a new compiler backend, it is helpful to perform a few simple compilations by hand to get an idea of what the expected output code should look like. This is also what my peers and I did at the start of our project. Figure 2 is a snippet of one of the Agda programs used for these compilations. Figure 3 shows the resulting by-hand-created Rust code. Since it is relatively easy for us humans to understand the goal of a code snippet, the result obtained from these manual conversions will be a lot more readable than the automatically compiled output.

After this initial experimentation, it was clear that most of the basic Agda features could be translated directly into valid Rust code. At this point, I decided to attempt to keep the generated Rust code as close to the initial Agda implementation and as humanly readable as possible. This approach would provide multiple benefits. It improves both the code's ability to be debugged and allows for easier FFI as the generated definitions would match the ones found in a typical Rust codebase. During this initial exploratory phase of the project, we also studied existing Agda compiler backends, such as the built-in GHC backend [The Agda Team 2022, GHC Backend] and the example Scheme backend [Cockx, Jesper 2022] provided for the project.

Next, we started implementing solutions for the conceptual differences between Rust and Agda. The main problems here were currying and laziness. A more detailed explanation of the problems and their solutions can be found in Section 4.

Most of the time during the project was spent on implementing the actual Agda compiler backend. The *agda2rust* compiler backend is written Haskell as this makes the integration with the existing Agda compiler easier. The compilation pipeline of the backend is split up into multiple phases, which will be discussed in more detail in Section 4.1.

```
postulate A : Set

id : A -> A
id x = x

data Bool : Set where
    true false : Bool

not : Bool -> Bool
not true = false
not false = true
```

Fig. 2. Boolean Agda example

```
fn id<A>(x: A) -> A {
    x
}

use Bool::*;
enum Bool {
    True(),
    False(),
}

fn not(x: Bool) -> Bool {
    match x {
        True() => False(),
        False() => True(),
    }
}
```

Fig. 3. Translated Agda source code

Throughout the development, we encountered quite a few implementation obstacles. This meant that it was not possible to implement all the required features due to a lack of time, but mainly because they could not be translated to Rust using the approach decided upon at the start of the project. Section 5 will discuss these obstacles in more detail.

Lastly, a simple proof of concept was built to show how Agda and Rust could be used side-by-side by integrating the *agda2rust* compiler into the Cargo build system. This allows Rust code to call functions and access constants defined in Agda.

4 IMPLEMENTATION

Integrating a new backend into the Agda compiler is relatively straightforward, as the code is structured modularly. The Agda compiler is exposed to the developer as a Haskell library which defines a set of hooks that allows the custom backend to influence the compilation process. After the Agda compiler has performed the static analysis and type-checking, an abstract representation of the code is passed to the respective backend. Next, the *agda2rust* backend will use this representation to emit valid Rust code. The goal of this project was thus to implement this last step.

4.1 Compilation Process

Like almost all modern compilers, the Agda compiler uses a pipelined architecture. The final output binary is created by passing the input source code through a set of transformations and checks. Adding support for a new target language or backend in the Agda compiler is relatively straightforward as we just replace the last step of the compilation process.

After the main Agda compiler has performed the type-checking and optimisations, the *agda2rust* backend receives an abstract version of the input source code called the treeless syntax [The Agda Team 2022, Treeless Syntax]. This treeless syntax is a simplified version of the original program. Its simplified nature makes the implementation of a new backend a lot easier as it contains far fewer features than the original language. This makes the treeless syntax more suitable for conversion into different target languages.

Before we can output the final Rust code, the *agda2rust* backend first performs a few transformations on this treeless syntax. Each transformation results in a new Intermediate

Representation (IR). In the *agda2rust* compiler, two different IRs are used: the High-level Intermediate Representation (HIR) (not to be confused with the inconveniently named Rust HIR) and Low-level Intermediate Representation (LIR). The HIR is a relatively straightforward translation of the Agda code into Rust concepts, while the LIR is a Haskell representation of a subset of the Rust Abstract Syntax Tree (AST). The implementation of these two IRs can be found in the GitHub repository[1]. These intermediate representations will be discussed in more detail in Sections 4.3 and 4.4. A complete overview of the compilation process can be seen in Figure 4.

Fig. 4. Compiler Overview

The two different IRs in the *agda2rust* backend are closely related but carry some significant differences. The HIR is a close representation of the original treeless syntax with some minor modifications, while the LIR is a subset of the Rust AST. During the compilation, three transformations have to happen: treeless syntax to HIR, HIR to LIR, and LIR to Rust code.

4.2 Type Representation

Before delving into the details about the IRs, we first need to understand how types are handled in *agda2rust*. Since one of the goals of the *agda2rust* backend is to provide FFI between Agda and Rust, it is necessary to keep the data types as close to traditional Rust code as possible, thus eliminating the need for converting types between the two languages. Luckily, Rust's enum types are powerful as each enum variant can also store additional data [The Rust Team 2022a, Defining an Enum]. Figures 5 and 6 show how this concept maps quite well to Agda's data types.

```
data Nat : Set where
  zero : Nat
  suc  : Nat → Nat
```

Fig. 5. Agda datatype

```
enum Nat {
  Zero(),
  Suc(Box<Nat>),
}
```

Fig. 6. A recursive Rust enum

The *Box<...>* (essentially a pointer to the heap) in Figure 6 is necessary as every enum in Rust needs to have a known size at compile time. Without the *Box*, we have an infinitely large datatype.

In most cases, the dependent types in data constructors are erased as part of the type-checking done by the Agda compiler. This leaves only the more conventional generic types behind, which will be handled as normal generic type parameters in the generated Rust code. This implementation, paired with the currying described in Section 4.5, comprises the complete Agda datatype implementation in *agda2rust*.

[1] *Hir.hs* and *Lir.hs* in https://github.com/HectorPeeters/agda2rust

4.3 HIR

The first transformation made by the *agda2rust* backend is converting the treeless syntax to HIR. This transformation only performs a handful of tasks mainly related to converting a functional language into an imperative one. The result of this transformation is a high-level view of the original program. For a more detailed overview, see the *agda2rust* repository[2].

As dependent and generic types are internally stored using De Bruijn indices [Berghofer and Urban 2007], we have to convert them into a more human-readable alternative. This is done by assigning each absolute index to a letter, starting at the letter A. As these will be converted to generic type parameters in the Rust code, they will also match the built-in naming conventions. Like the types, we also have to assign names to variables used throughout the code. This works similarly by allocating names starting at the letter a.

A different problem occurs when working with function names. Since identifiers in Agda can consist of almost any Unicode character, we have to correctly convert those, as Rust is not so liberal with its variable naming [The Rust Team 2022c, Identifiers]. A naive approach which works well enough for this research compiler consists of replacing every invalid character with its hexadecimal representation. Since Rust identifier names cannot start with a number, we also prepend the letter x. While this approach could result in name clashes, they are unlikely to happen, as the developer would have to write identifier names containing hexadecimal number literals.

This conversion to HIR also inserts dereference operators into the IR where necessary. As most variables in Agda are only used once, the decision was made to move values instead of passing them around as references. As a move essentially transfers ownership of a variable, it does not come with a performance penalty as these cases get optimised away by the Rust compiler. Moving values aligns more with the functional programming style and simplifies the code generation as we only have to insert dereference operators when necessary. Along with the dereference operators, the compiler also has to insert clone operations throughout the code. This is required for the currying, as explained in Section 4.5.

After performing these transformations, the HIR is complete and gets passed to the next stage of the compiler. The formatted HIR of the example code from Figure 2 is shown in Figure 7. Note that the vertical bars denote a lambda function where the letter between the bars is the argument name.

```
fn id(A, A) =
  |a| a

data Bool (
  ztrue: Bool,
  zfalse: Bool
)

fn not(Bool, Bool) =
  |a| match a
    Bool::ztrue() -> zfalse()
    Bool::zfalse() -> ztrue()
```

Fig. 7. Agda Sample HIR

[2] *toRust.hs* in https://github.com/HectorPeeters/agda2rust

4.4 LIR

As the HIR code still contains concepts that cannot be converted into Rust or that will not work in the Rust implementation, we convert it into the LIR. While this LIR is quite similar to the HIR, it can immediately be translated to Rust source code without performing any other transformations.

The main task of the conversion from HIR to LIR is generating functions that can be curried and making sure the code can be lazily evaluated. Both of these topics will be explained in more detail in the following two sections. For the full implementation of the conversion, please see the *agda2rust* repository[3].

The final result of the compilation of the example source from Figure 2 after converting the LIR to Rust source code can be found in Figure 8. The actual output also includes some feature flags and linting annotations, but these have been excluded to keep the code as readable and compact as possible. This example also shows that the generated code is by far not optimal and generates some unnecessary functions and type aliases. Luckily, most of these do not affect the performance as they get inlined by the Rust compiler.

```
type id0<A: Clone> = impl FnOnce(Lazy<A>) -> A;
fn id<A: Clone>() -> id0<A> {
  move |a| a.clone()
}

#[derive(Debug, Clone)]
enum Bool{
    ztrue(),
    zfalse(),
}

fn ztrue() -> Bool {
  Bool::ztrue()
}

fn zfalse() -> Bool {
  Bool::zfalse()
}

type not0 = impl FnOnce(Lazy<Bool>) -> Bool;
fn not() -> not0 {
  move |a| match a.clone() {
    Bool::ztrue() => zfalse(),
    Bool::zfalse() => ztrue(),
    _ => unreachable!(),
  }
}
```

Fig. 8. Agda sample code compiled to Rust

[3] *HirToLir.hs* in https://github.com/HectorPeeters/agda2rust

4.5 Currying

While Rust already supports many of Agda's features like pattern matching and algebraic data types, as seen in Figures 2 and 3, there are still some functional programming specific features that Rust does not support.

One of the most prominent features of functional programming languages is the support for function currying. A curried function is a function that takes only one argument and returns either the return value or another function with one argument. We could try writing a curried function in Rust as a function that returns another function. As Rust supports closures and the corresponding *Fn* and *FnOnce* [The Rust Team 2022a, Closures], we can use these to represent the return type of our function, which can be seen in Figure 9.

```
fn add(x: u32) -> impl FnOnce(u32) -> impl FnOnce(u32) -> u32 {
    move |x| move |y| x + y
}
```

Fig. 9. Ideal curried function

This implementation would be ideal as it is relatively simple to generate and allows us to write the body of our function without adding any unnecessary complexity. However, this code snippet does not compile as `impl FnOnce` is only allowed in function return types and not nested in another `impl FnOnce`. One solution is to box [The Rust Team 2022a, Smart Pointers] the nested `impl FnOnce` types, which makes them heap-allocated, as shown in Figure 10.

```
fn add(x: u32) -> impl FnOnce(u32) -> Box<dyn FnOnce(u32) -> u32> {
    move |x| Box::new(move |y| x + y)
    //            ^ this creates a heap allocation
}
```

Fig. 10. Heap allocated curried function

While this approach works, it does make the generated code about twenty times slower and makes the generation of the function body more complex. Luckily, an experimental feature[4] in the current Rust version (1.61.0) allows us to create a type alias and compile the initial snippet without heap allocations. The approach seen in Figure 11 will be applied to all the generated functions and the constructors for data types.

```
#![feature(type_alias_impl_trait)]

type TestCurry1 = impl FnOnce(u32) -> u32;

fn test(x: u32) -> impl FnOnce(u32) -> TestCurry1 {
    move |x| move |y| x + y
}
```

Fig. 11. Final curried function

[4]https://rust-lang.github.io/rfcs/2515-type_alias_impl_trait.html

To integrate currying into the existing compiler, we have to ensure that both normal functions and data constructors can be curried. We can achieve the latter by creating a curried function for every datatype variant, as shown in Figure 12. To simplify the code generation process, the code emitted wraps every non-zero-argument function in a zero-argument function, which returns the original function. This unifies the implementation for constants, zero-argument functions, and multiple argument functions.

```
enum Nat {
  Zero(),
  Suc(Box<Nat>),
}

fn zero() -> Nat {
  Nat::Zero()
}

fn suc() -> impl FnOnce(Nat) -> Nat {
  move |x| Nat::Suc(Box::new(x))
}
```

Fig. 12. Curried Nat constructor

4.6 Laziness

The second main conceptual difference between Agda and Rust is the evaluation strategy. Agda employs lazy evaluation, while Rust uses strict evaluation. This requires us to convert the lazy code into a strict alternative. The most conventional implementation uses *thunks* [Reem, J. and Silva, A. and Wilson, H. 2015], a data structure that acts as a wrapper around a to-be-evaluated function parameter. Only when the value of a thunk is being used does it get evaluated. The result is cached and reused if the thunk is accessed elsewhere. Thunks essentially ensure optimal parameter evaluation as they are only evaluated when needed and evaluated at most once. This is not the case for *call-by-name* which evaluates all parameters before calling a function, or *call-by-value*, which evaluates an argument every time it is used.

The thunk implementation in *agda2rust* is an adaptation of the implementation in the Rust standard library [The Rust Team 2022d, std::lazy::Lazy]. This implementation was chosen as it does not use any unsafe code, ensuring all the safety features provided by Rust. Examples of unsafe operations in Rust include dereferencing a raw pointer, accessing a union field, or calling an unsafe function. Due to an inconvenience in the way Rust handles closures, not all function arguments can be converted to thunks. When the value of a function argument borrows other variables in the same scope, it cannot be thunked. A solution would be to separately store the captured variables inside the thunk as an environment, but due to the limited time span of this research project, this feature was not implemented.

The current implementation, however, comes at a performance cost as evaluating the standard library thunk results in a reference to the internally stored value. Since the code generated by *agda2rust* relies on moving values around and not passing references, we have to eliminate the reference that is being returned. You convert a reference type to a standard type in Rust by cloning that value which performs a memory copy. This heavily impacts performance and could be avoided with an alternative implementation of the thunk type.

A different solution could be based on the *rust-lazy*[5] library, which returns a non-reference type when evaluated. It achieves this by performing unsafe operations, which would make a big part of the codebase unsafe. Having the thunk implementation be left as a choice to the user could be an adequate solution. The choice then lies between safety or performance.

4.7 Optimisations

Natural numbers in Adga are represented using the Peano number system. This number system defines a *zero* and a relation that defines the successor of a number. With these two simple concepts, we can define all possible natural numbers. The number three can then be defined as *suc suc suc zero*.

These Peano numbers, however, are inefficient for performing any actual computations. That is why the Agda compiler also supports generating conventional or *primitive* integer types and operations. These are converted directly into integer literals and operations in Rust and thus perform as fast as native Rust code.

The same optimisations can be applied to boolean types, but these are not present in the current version of the compiler.

5 RESULTS

In this section, we will take a closer look at the results. First, we will analyse the performance and compare the results to existing backends. Next, limitations and missing features will be discussed. Lastly, we will analyse the interoperability between Agda and Rust.

5.1 Benchmarks

All the benchmarks have been run using *hyperfine*[6], a terminal-based benchmarking tool. The setup used in this project first performs ten warmup rounds to remove any cache-related delays. Then *hyperfine* runs at least ten runs for each input and keeps performing benchmarks for at least ten seconds per input. This results in many samples for smaller inputs as they are faster and fewer samples for more significant inputs.

The main benchmark ran during this research was a test where a Peano number gets constructed using a recursive data structure and then consumed by subtracting one until zero is reached again. The natural number optimisations have been disabled to limit the benchmark to only function calls and datatype construction and destruction. This benchmark was executed using a stack of approximately eight gigabytes since the *agda2rust* backend does not support tail-call optimisations [Schwaighofer 2009]. This prevents the stack from growing too big when executing tail-recursive functions. Without this optimisation, every recursive call adds a new stack frame onto the stack, which can then easily overflow. Tail-call optimised functions reuse the previous stack frame, which thus prevents the stack from growing too quickly.

As seen in Figure 13, the two executables generated using the *agda2rust* are the worst performing of all the backends. The difference between Rust and Rust-optimal is that Rust-optimal is written how a human would write Rust code. This also removes function currying. Please note that on the graphs, Rust and Rust-optimal have almost identical performance and are thus indistinguishable. Since their results closely match, we can assume that the Rust compiler correctly optimises all the currying-related code. Zooming in on the graph as in Figure 14, we can see that the startup cost for the strict Rust versions is significantly

[5]https://github.com/reem/rust-lazy
[6]https://github.com/sharkdp/hyperfine

lower than the Scheme backend by a factor of about 35. Only for sufficiently small inputs can it outperform the Haskell backend.

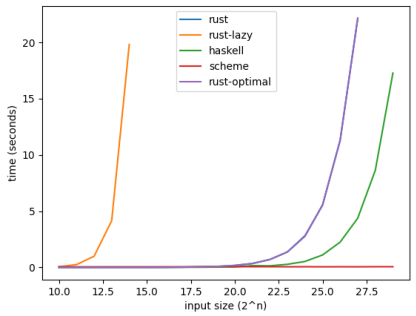

Fig. 13. Peano benchmark

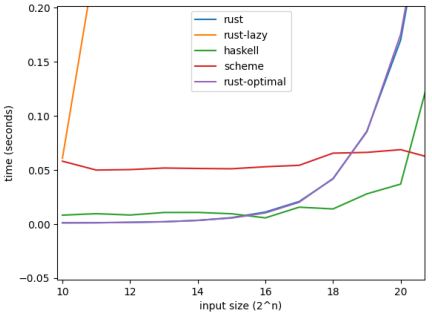

Fig. 14. Peano benchmark (zoomed in)

Without knowing all the details of the inner workings of the Rust compiler, we can only assume that the Rust code generally performs worse than the Haskell code purely because the Haskell compiler would be optimised for a more functional code style while Rust is better suited for imperative programming.

Laziness also adds a relatively large performance overhead, as the performance graph shows. The execution time is several magnitudes larger than the strict version of the code. This performance regression is present due to the current implementation of the thunk values as discussed in Section 4.5. An alternative and unsafe implementation would undoubtedly perform many times better. With more time on our hands, this implementation could be added as a special feature flag since the Rust code would not be completely safe anymore.

5.2 Limitations

The current currying implementation generates a nested set of closures as the return type of a function which allows functions to be partially applied. However, when one of the arguments is a function itself, the generated code contains two nested function types. This is not allowed in the current version of the Rust compiler and thus will not compile. Due to time constraints, this bug was not resolved on time, so the final *agda2rust* backend does not support higher-order functions.

Secondly, in Agda, all types have a common parent class called *Set*. This abstraction essentially blurs the line between types and values. Unfortunately, Rust does not natively support this feature as types and values are seen as two different concepts. The more straightforward translation of the Agda source to Rust used in this project can cause problems in some situations. Most of the time, types which are used as values are erased so we do not have to generate code for them, but in some situations, like in the example working with a type universe [Chapman et al. 2010] shown in Figure 15, this is not the case. The function $[\![\_]\!]$ takes in a normal value and returns a *Set* or type. When the *agda2rust* backend encounters this situation, it emits incorrect code. The most elegant way to solve this limitation is to switch from the current code generation to a more interpreter-styled approach where Agda values and types are not stored as literal Rust values and types but stored in an internal array combining the two. Passing around types and values can then be done using the index into

the array instead of an instance of the type, which makes types and values indistinguishable.

```
open import Agda.Builtin.Bool
open import Agda.Builtin.Nat

data U : Set where
  nat : U
  bool : U

⟦_⟧ : U → Set
⟦ nat ⟧ = Nat
⟦ bool ⟧ = Bool

f : (c : U) → ⟦ c ⟧ → ⟦ c ⟧
f nat      zero     = zero
f nat      (suc x)  = x
f bool     false    = true
f bool     true     = false

test  = f nat 42
```

Fig. 15. Agda universe example code

5.3 Interoperability

One of the project's goals was to combine Agda and Rust and use them in the same project. Since the *agda2rust* compiler tries to translate the Agda code into the most straightforward Rust counterpart, the output looks quite similar to how a normal human would write it. This property of the compiler makes it relatively easy to integrate Agda code into an existing Rust repository.

As Cargo, the Rust build system can be quite easily extended; a custom build script can be used to compile the Agda code and include the output into the main codebase. Compiling Agda and Rust is as easy as running *cargo run*. A simple example can be found in Figure 16. This functionality allows an algorithm to be implemented and proven in Agda, which can then be used in a larger Rust codebase. For the complete setup, including the build script, see the *agda_rust_interop* GitHub repository[7].

6 RESPONSIBLE RESEARCH

The *agda2rust* compiler acts as a backend for the existing Agda compiler. All the data passed to *agda2rust* has already been checked, and unnecessary information has been removed. As the backend is entirely deterministic, it cannot result in bias or incorrect statistics. The repository containing the *agda2rust* source code is open source and available on GitHub[8] for reproducibility purposes.

[7]https://github.com/HectorPeeters/agda_rust_interop
[8]https://github.com/HectorPeeters/agda2rust

```rust
#![feature(type_alias_impl_trait)]

// Include the compiled Agda file
include!(concat!(env!("OUT_DIR"), "/Nat.rs"))

fn main() {
  let result: Nat = plus()(two())(three());
  println!("{:?}", result);
}
```

Fig. 16. Using Agda functions and data types in Rust

7 CONCLUSION

This project aimed to determine whether Rust was a feasible target language for extracting code from Agda, and in this paper, we have shown that this depends on what it will be used for.

With the current implementation, *agda2rust* would not be a good alternative backend if performance is essential, as the generated code is slower than the built-in Haskell backend. Implementing additional optimisations might improve this situation, but it is likely that Rust is just not designed for a purely functional programming style. Moreover, the strong typing built into the language makes code generation for Rust much harder than for languages with a more flexible type system. The strict typing in the Rust language is also not required as the code has already been type-checked by the Agda compiler.

Furthermore, there are some design issues that show that the backend will not be fully feature-complete without a drastic redesign. The way *agda2rust* is implemented does not support using types as values, which is possible in Agda.

Improving upon the current implementation can be done both in terms of optimisations to bring the runtime performance closer to the current Haskell backend or by implementing missing features like higher-order functions.

The area where *agda2rust* really shines is interoperability. Due to the nature of the generated code, calling Agda code from Rust is trivial. The integration with the Cargo build system allows for an elegant and efficient workflow when combining Agda and Rust.

8 FUTURE WORK

Since this research project was conducted in a relatively short period, the final implementation of the *agda2rust* compiler is far from complete. It currently contains a few known bugs and is still missing some features and optimisations.

The compiler's most significant feature currently missing is the support for higher-order functions. This would allow the compiler to accept expressions like *(filter even xs)*. While the function generation code supports arguments which are functions, there is still an unresolved bug in the implementation, which results in the incorrect arguments being used inside the body of the function. Resolving this problem should allow the backend to compile and use higher-order functions.

Due to time constraints, the *agda2rust* backend is still missing some straightforward optimisations. These include supporting primitive booleans and removing erased types from the generated functions. Primitive booleans are comparable to the primitive numbers discussed in Section 4.7. It is worth noting that this optimisation would probably not gain that much

performance as simple enums (like *Bool*) in Rust are stored as a byte just like the native *bool* type.

Another optimisation which would improve the execution speed is removing erased arguments from the generated code. When arguments are not required for execution, they get replaced using an erased argument. The backend currently converts these into a unit type [The Rust Team 2022c, Tuple Types]. This causes problems as some erased arguments might get a numerical or boolean value assigned when a function is called, which would then fail to compile. Leaving out these arguments would fix this bug and optimise the overall function execution speed.

The last optimisation which would greatly improve the *agda2rust* compiler is tail-call optimisation, as explained in Section 5.1. For implementing this optimisation, it would be possible to use an existing library like *tramp-rs*[9] or *tco*[10].

Finally, other systems languages with a more flexible type system might be worth considering, such as *Zig* [Zig Software Foundation 2022] or *Nim* [The Nim Contributors 2022]. As the Agda code is already type-checked, the language we are targetting does not need to be as strict with its types.

9 RELATED WORK

Code extraction is not a new concept and is widely used in modern compilers. One might even argue that every compiler performs code extraction by outputting assembly instructions or emitting bytecode like the Java HotSpot compiler outputs JVM bytecode [Kotzmann et al. 2008].

Unfortunately, the amount of compilers which extract a strongly typed imperative systems language from a dependently typed functional language is still quite limited. The two closest related projects are the current JavaScript backend for Agda [The Agda Team 2022, JavaScript Backend] and the CertiCoq [Anand et al. 2017] compiler for the Coq language targetting a subset of C. While the former performs the conversion from a functional to a systems language, it still targets an interpreted language which has some performance drawbacks. The CertiCoq compiler, on the other hand, does compile to a native executable, but it was built with a different goal in mind. The goal of the CertiCoq project was to create a certified compiler for the Coq language using the CompCert C compiler [Leroy et al. 2016].

Since this project aims to investigate whether Rust would serve as a viable target language, both in terms of performance and interoperability aspects, the two previously described compilers do not serve as complete solutions for the problem in question. Only the JavaScript backend for the current Agda compiler has been consulted during this research, as it solves the same problem of converting a functional language to an imperative one.

REFERENCES

Abhishek Anand, Andrew Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Belanger, Matthieu Sozeau, and Matthew Weaver. 2017. CertiCoq: A verified compiler for Coq. In *The third international workshop on Coq for programming languages (CoqPL)*.

Stefan Berghofer and Christian Urban. 2007. A Head-to-Head Comparison of de Bruijn Indices and Names. *Electronic Notes in Theoretical Computer Science* 174, 5 (2007), 53–67. https://doi.org/10.1016/j.entcs.2007.01.018 Proceedings of the First International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP 2006).

[9]https://gitlab.com/bzim/trampoline-rs

[10]https://github.com/samsieber/tco

Ana Bove, Peter Dybjer, and Ulf Norell. 2009. A brief overview of Agda–a functional language with dependent types. In *International Conference on Theorem Proving in Higher Order Logics*. Springer, 73–78.

James Chapman, Pierre-Évariste Dagand, Conor McBride, and Peter Morris. 2010. The gentle art of levitation. *ACM Sigplan Notices* 45, 9 (2010), 3–14.

Cockx, Jesper. 2022. agda2scheme. https://github.com/jespercockx/agda2scheme.

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2021. Safe systems programming in Rust. *Commun. ACM* 64, 4 (2021), 144–152.

Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. 2008. Design of the Java HotSpot™ Client Compiler for Java 6. *ACM Trans. Archit. Code Optim.* 5, 1, Article 7 (may 2008), 32 pages. https://doi.org/10.1145/1369396.1370017

Chris Arthur Lattner. 2002. *LLVM: An infrastructure for multi-stage optimization*. Ph. D. Dissertation. University of Illinois at Urbana-Champaign.

Woo Hyong Lee and Morris Chang. 2002. A study of dynamic memory management in C++ programs. *Computer Languages, Systems & Structures* 28, 3 (2002), 237–272. https://doi.org/10.1016/S0096-0551(02)00015-2

Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. 2016. CompCert-a formally verified optimizing compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*.

Pierre Letouzey. 2008. Extraction in Coq: An Overview. In *Logic and Theory of Algorithms*. Lecture Notes in Computer Science, Vol. 5028. Springer, Krakow, 359–369.

Reem, J. and Silva, A. and Wilson, H. 2015. rust-lazy. https://github.com/reem/rust-lazy.

Arnold Schwaighofer. 2009. Tail Call Optimization in the Java HotSpot ™ VM. , 6-19 pages.

The Agda Team. 2022. The Agda Documentation. https://agda.readthedocs.io/en/v2.6.2.1/.

The Nim Contributors. 2022. Nim Website. https://nim-lang.org/.

The NodeJS Team. 2022. An Introduction to the NPM Package Manager. https://nodejs.dev/learn/an-introduction-to-the-npm-package-manager.

The Rust Team. 2022a. The Official Rust Book. https://doc.rust-lang.org/book/.

The Rust Team. 2022b. Rust Programming Language Website. https://www.rust-lang.org/.

The Rust Team. 2022c. The Rust Reference. https://doc.rust-lang.org/stable/reference/.

The Rust Team. 2022d. The Rust Standard Library Documentation. https://doc.rust-lang.org/std/index.html.

David A Turner. 2004. Total Functional Programming. *J. Univers. Comput. Sci.* 10, 7 (2004), 751–768.

Aaron Weiss, Daniel Patterson, Nicholas D. Matsakis, and Amal Ahmed. 2019. Oxide: The Essence of Rust. *CoRR* abs/1903.00982 (2019). arXiv:1903.00982 http://arxiv.org/abs/1903.00982

Zig Software Foundation. 2022. Zig Website. https://ziglang.org/.