UNIVERSITAT DE BARCELONA

Treball final de grau

GRAU D'ENGINYERIA INFORMÀTICA

Facultat de Matemàtiques i Informàtica
Universitat de Barcelona

Design and implementation of a C to Rust transcompiler

Autor: Lluís Alonso Jané

Director:	**Dr. Lluís Garrido Ostermann**
Realitzat a:	**Departament de Matemàtiques i Informàtica**
Barcelona,	**29 de gener de 2018**

Contents

Abstract

A transcompiler or source to source compiler is a type of compiler that translates source code from a programming language at a level of abstraction to another programming language at the same level, contrary to a traditional compiler which translates from a level of abstraction to a lower one.

This project is an implementation of a transcompiler from C to Rust, a relatively recent systems programming language that operates at a similar level of abstraction but with a focus on safety in regards to memory.

The scope of this project is limited, but it should work on most small C programs that do not use advanced features (like thorough memory management or pointer arithmetic) or that work with multiple threads.

Resum

Un transcompilador, o compilador de font a font, és un tipus de compilador que tradueix codi font d'un llenguatge de programació a un cert nivell d'abstracció a un altre llenguatge al mateix nivell, a diferència d'un compilador tradicional que tradueix d'un cert nivell d'abstracció a un d'inferior.

Aquest treball és una implementació d'un transcompilador de C a Rust, un llenguatge de programació de sistemes relativament recent que opera a un nivell d'abstracció similar però amb un èmfasi a la seguretat en termes de memòria.

L'abast del treball és limitat, però hauria de funcionar en la majoria de programes petits en C que no utilitzin característiques avançades del llenguatge (com extensiva gestió de memòria o aritmètica de punters) o que siguin multifil.

Resumen

Un transcompilador, o compilador de fuente a fuente, es un tipo de compilador que traduce código fuente de un lenguaje de programación a un nivel de abstracción determinado a otro lenguaje al mismo nivel, a diferencia de un compilador tradicional que traduce de un cierto nivel de abstracción a otro de inferior.

Este trabajo es una implementación de un transcompilador de C a Rust, un lenguaje de programación de sistemas relativamente nuevo que opera en un nivel de abstracción similar pero con un énfasis en la seguridad en términos de memoria.

El alcance de este trabajo es limitado, pero debería funcionar en la mayoría de programas pequeños en C que no utilicen características avanzadas del lenguaje (como extensiva gestión de memoria o aritmética de punteros) o que sean multihilo.

1. Introduction

Transcompilers are a type of compiler that usually translate source code between languages at the same level of abstraction.

The first compilers were designed and developed during the 1950s, mostly to provide a language at a higher level of abstraction than assembly code[1].

As programming languages became more feature-rich and computer architectures became more complex, the use of compilers saw a rise in usage, while less and less code was written directly in assembly.

While the first transcompilers were created to port codebases from a specific platform to another, a use case still present nowadays, there have been new uses for such type of compilers, like extending languages or creating new languages that compile to existing ones to increase its ease of use or eliminate potential errors when using the existing ones.

That said, while widely used (specially with the latter use case), there is little documentation on what differentiates them to traditional compilers or their peculiarities, which make for a good subject to study.

This report goes over some of the uses and peculiarities of transcompilers, as well as presenting one such transcompiler and its design with the intent of further studying this type of software.

1.1. Motivation

The motivation behind this project lays in the will to learn more about transcompilers, and compilers in general. While much is known about compiler design, it is a subject that does not attract as much research as new technologies like artificial intelligence or computer vision[2]. Additionally, most compilers still use the base structure that they were using when they first appeared, and most of its research goes to further improve the speeds of the compilation process and the optimization of its output and not its design or structure. Therefore, it is an interesting subject to study as it can be seen why the current day structures work and have a solid base.

Another reason this was the chosen project was the ability to work with a relatively new language, and see its differences in design with a more traditional language like C. In this case, the focal point of interest was that Rust as a language aims to provide a similar feature set to that of C while removing some of its arguably worst parts, like undefined behaviour or careless memory management.

[1] The first commercial compiler was IBM's FORTRAN compiler[1].

[2] According to *arxiv.org*, the amount of papers released in the month of January, 2018, in each of those two subjects alone were more than eight times of those released regarding programming languages, let alone compilers.

1.2. Organization

This report is divided by the following sections:

1.- Introduction: The current section.

2.- Background: This section contains several briefings about core concepts that have been used during the creation of this project.

3.- Architecture and design: This section contains the explanation of how the project was structured, as well as how it works internally when being executed.

4.- Tests and examples: This section explains the methodology used when developing the project, as well as a list of features that work and a list of ones that do not. Additionally, there are some examples at the end that show the results obtained when using the project.

5.- Conclusions: This section contains a self-evaluation of the project, as well as future lines to expand and improve the project.

2. Background

2.1. *Rust*

Rust is a systems programming language sponsored by Mozilla Research, which describes it as a "safe, concurrent, practical language"[2]. The *Rust* project was created because other languages at this level of abstraction and efficiency, such as C and C++, are unsatisfactory. In particular:

- There is too little attention paid to safety.
- They have poor concurrency support.
- There is a lack of practical affordances.
- They offer limited control over resources.

Rust exists as an alternative that provides both efficient code and a comfortable level of abstraction, while improving on all four of these points. [2]

This design philosophy has led the language to have a feature set with an emphasis on safety, control of memory layout and concurrency. Additionally, performance of idiomatic Rust is comparable to C++.[2]

One of the strongest points of Rust is that, while using strictly `safe` Rust, one should never see any kind of undefined behaviour. This is achieved via strict controls at compile time: where a C compiler would maybe throw a warning, the Rust compiler will throw an error and prevent compiling a binary which does not have its behaviour explicitly stated in the source code. An example of this would be dereferencing a raw pointer: because the compiler does not know if the program has access to the memory address the pointer corresponds to, the Rust compiler will throw an error when trying to do this, whereas the C compiler will not.

While the previous is true, in Rust the programmer can do things that the compiler would consider undefined behaviour by using the `unsafe` keyword. Everything in the `unsafe` block will not throw errors related to undefined behaviour (normal errors still apply) and the compiler will assume that the programmer knows what it will do. The following is the list of things that the Rust compiler considers undefined behaviour:

- Dereferencing null, dangling or unaligned pointers.
- Reading uninitialized memory.
- Unwinding into another language
- Producing invalid primitive values (like a `bool` that is not 0 or 1).
- Causing a data race[3]

As it can be seen, the Rust language is limited when it comes to undefined behaviour and quite permissive as it considers "safe" things like having a deadlock, a race condition, leak memory or overflow integers, among others.

With that said, this projects aims to provide (almost always) `safe` Rust code.

2.2. Compilers

A compiler is computer software that transforms code written in one programming language into another. The name compiler is mainly used for those programs that translate source code from a high-level programming language to a lower level language.

A compiler is divided in two big parts: analysis and code generation. The analysis part is always the same:

- Lexical analysis: Initial reading of the source code. The source code is read and divided into *tokens*, each of which corresponds to a symbol in the programming language.
- Syntax analysis: This phase takes the list of *tokens* produced in the previous phase and arranges these in a tree-structure (knows as *abstract syntax tree* or AST) that reflects the structure os the program. This phase is often called *parsing*.
- Semantic analysis: This phase analyses syntax tree to determine if the program violates certain consistency requirements, *e.g.*, if a variable is used but not declared in C. The previous tree is not modified.[4]

In some cases a preprocessor is present, which prepares the source text before entering the first phase. As an example, *gcc*'s preprocessor merges the source code of the file to compile with the headers found in the `#include`s of the main file, as well as processing the `#define` statements and making the proper substitutions in the code.

The code generation part is usually compound of:

- Intermediate code generation: The program (an AST in this phase) is translated to a machine-independent intermediate language, usually in the form of tables. This is used to ease the compiling task towards multiple targets, *i.e.*, different CPU architectures.
- Code optimization: In this phase the code found in the intermediate language tables is optimized, but still keeping it in this language. An example would be eliminating unused variables or functions that are never called.
- Code generation: The final code is produced from the translation tables. In a traditional compiler, the result would be the source program translated to assembly language ready to be executed.

While the previous is true, it is not uncommon for a compiler to go straight from the abstract syntax tree to the final code or to not directly optimize code, like the Java compiler (which does not optimize the source code when translating to *bytecode*). Additionally, there are compilers that do another *linking* phase where the multiple parts of a program are merged after compiling each of them individually.

In transcompilers' case, it is usually not necessary the intermediate code generation as they compile always to another programming language (a single compilation target). This project works this way, translating the abstract syntax tree directly to the final code in Rust.

2.3. Transcompilers

A transcompiler or source-to-source compiler is a type of compiler that takes the source code of a program written in a programming language as input and outputs the equivalent code in another programming language.

A transcompiler translates between two languages that operate at the same level of abstraction, while a traditional compiler translates from a high-level language to another in a lower level.

2.3.1. Previous work

The first source to source compiler was developed in 1981. It translated .ASM source code for the Intel 8080 processor into .A86 source code for the Intel 8086[5]. Since then, a multitude of transcompilers have been developed, specially for internal use in the enterprise sector, to port codebases from a programming language to another. Current day examples of such use would be Emscripten[8], which compiles C and C++ to JavaScript (in this case it works from a lower level language to one higher) so standalone applications can be easily ported and embedded in the web, or BaCon[9], which converts BASIC code to C.

While this use case is still present, the increasing popularity of languages such as Python and JavaScript, due to their important presence on the Internet and ease of use, has brought a new use for transcompilers: creating new languages that compile to the widely used ones while bringing new features. Examples of that would be CoffeeScript[10] and TypeScript[11], both of which compile to JavaScript[3] while providing new features (like type checking and build systems) or preventing common errors while providing optimizations.

Although the later use of transcompiler is readily available as well as its source, the former is usually not. This project works more like a traditional transcompiler, in the sense its use case is to translate a codebase from a language to another, with manual revision afterwards to ensure correct functionality and optimization.

[3] CoffeeScript is its own language that compiles to JavaScript, while TypeScript considers itself a superset of JavaScript that compiles to it.

2.3.2. Tools used

The main tool used is the `pycparser`[6] library, which is a parser for the C language, written in Python. This library does the first three phases of the transcompiling process (lexical, syntactical and semantic analysis).

This library uses, in turn, another: `PLY`[7], which is an implementation of `lex` and `yacc` parsing tools in Python. Specifically, it is a general purpose L-R parser.

`pycparser` uses `PLY` to do the *parsing*, the former providing the later the C language grammar provided in Annex A of the C99 standard (ISO/IEC 9899). Then, `pycparser` generates the AST corresponding to the original source code.

3. Architecture and design

3.1. Architecture

The project is structured akin to a compiler, see figure 1:

- Preprocessor: This block processes, if requested via the command line option `-d`, the compiler directives `#define`, creating its equivalent definitions in Rust. Afterwards it creates a version of the input source file without the `#include` directives that are part of a library (by standard defined as `#include <filename>`) to prevent problems with standard library parsing from `pycparser`. If requested via the command line option `-i`, the remaining `#include`s will also be removed.

  Once the file to parse has been prepared, it creates the file which describes the dependencies to the Rust compiler (`Cargo.toml`) using the user-set file `dependencies.json` (for more information on this files, see section 3.4).

  Finally, it prepares the dependency statements to be added in the source file (similar to the `#include` lines in C).

- `pycparser`: The library receives as input the source file processed by the previous block, passes it through the `gcc` preprocessor (which at this point only removes comments and processes the `#define`s found in the source code), and it parses it, generating with it the abstract syntax tree from the original source code.

- Code generator: The final code in Rust is generated from the abstract syntax tree created in the previous block.

This blocks are called and managed by the main module of the project, `c-to-rs.py`, which is also the responsible for all file input/output as well as managing command line options.

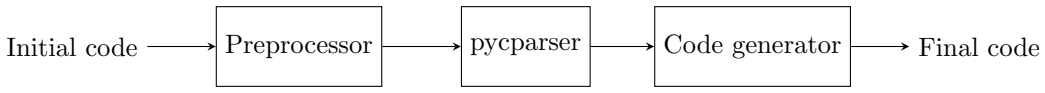

Figure 1: Block diagram of the project

3.1.1. Work done and usage

The work done on this project was focused on the following, ordered by amount of work:

- Code generator: The entire implementation of the code generator was part of the project, and where the most amount of time was put in.
- Preprocessor: Most of the preprocessing work done by the program was implemented for this project, with a small amount of work done by the C preprocessor.
- Main module: The assembly of all the parts to make the transcompiler work was also part of the project. This includes the integration of the previous parts as well as the usage of the `pycparser` library and the customization options.

Usage

To transcompile a file, the following command will do it as long as Python 2.6 or higher and `pycparser` are installed:

```
>python c-to-rs.py {file to transcompile}
```

This will execute the program and output the transcompiled version of the source file in the command to the `out/src` folder. To run said program (as long as Rust is installed), the user must go to the `out` folder and run: `cargo run`. The following are the command line options available with the transcompiler:

- `-h`: Prints the help of the transcompiler to the terminal.
- `-ni`: This option will remove all the `#include`s (even those that are not a part of a library), making the program transcompile exclusively the source file indicated in the command. This should be used if the user wants to create a similar file structure in the Rust program akin to that in the C one.
- `-v`: Both the *abstract syntax tree* and the final source code will be printed to the terminal when the transcompilation process finishes.
- `-d`: The transcompiler will create an equivalent to each of the `#define`s in the source file that are constant declarations (will not work with macros).

3.2. Analysis of a transcompilation example

Due to the complexity of the transcompiler, it is simpler to explain how it functions via an example. Given the following source code as input to the program:

```
#include <stdio.h>

void print_n(int n)
{
    for (int i = 0; i < n; i++)
        printf("Iteracio %d\n", i);
}

int add_one(int n)
{
    return ++n;
}

void main()
{
    int n = 10;
    print_n(n);
    printf("\nPausa\n\n");
    print_n(add_one(n));
}
```

Listing 1: Example source code to transcompile

Preprocessor

When entering the preprocessor, the first `#include` is removed, and the resulting file is passed on to the parser. Additionally, it generates the file `Cargo.toml` with the building options and the dependencies described in `dependencies.json`, in this case the example no dependencies will be introduced into `Cargo.toml` as the example dependencies found in said false have its `use` flag to `false`.

It also prepares the statements to include the libraries in the source file (equivalent to the `#includes` in C).

`pycparser`

The temporary file generated from the previous block is parsed by the library, generating an abstract syntax tree of the original source code.

An abstract syntax tree is the result of taking the *tokens* resulting from the initial *parsing* (also done by `pycparser`) and recombining them into a data structure that reflects the actual structure of the original source code.

The following is the abstract syntax tree of the example with its custom nodes:

```
FileAST:
  FuncDef:
    Decl: print_n
      FuncDecl:
        ParamList:
          Decl: n
            TypeDecl: n
              IdentifierType: ['int']
        TypeDecl: print_n
          IdentifierType: ['void']
    Compound:
      For:
        DeclList:
          Decl: i
            TypeDecl: i
              IdentifierType: ['int']
            Constant: int, 0
        BinaryOp: <
          ID: i
          ID: n
        UnaryOp: p++
          ID: i
        FuncCall:
          ID: printf
          ExprList:
            Constant: string, "Iteracio %d\n"
            ID: i
  FuncDef:
    Decl: add_one
      FuncDecl:
        ParamList:
          Decl: n
            TypeDecl: n
              IdentifierType: ['int']
        TypeDecl: add_one
          IdentifierType: ['int']
    Compound:
      UnaryOp: p++
        ID: n
      Return:
        ID: n
```

```
FuncDef:
  Decl: main
    FuncDecl:
      TypeDecl: main
        IdentifierType: ['void']
  Compound:
    Decl: n
      TypeDecl: n
        IdentifierType: ['int']
      Constant: int, 10
    FuncCall:
      ID: print_n
      ExprList:
        ID: n
    FuncCall:
      ID: printf
      ExprList:
        Constant: string, "\nPausa\n\n"
    FuncCall:
      ID: print_n
      ExprList:
        FuncCall:
          ID: add_one
          ExprList:
            ID: n
```

Listing 2: Abstract syntax tree of the C program. Every line corresponds to a node, with the first word being the type of the node and the rest being parameters of it. If a node is indented in relation to its predecessor it means it is the child node of said predecessor.

Code generator

This final block takes the previous tree, traverses it and generates the final Rust code. When visiting a node, that node is the responsible of taking the result of its children nodes, modify it if needed, and merging it. Finally, it adds the dependency statements generated by the preprocessor and appends them in the beginning of the text.

From the previous tree, the resulting code is as follows:

```
fn print_n(mut n: i32) {
    {
        let mut i: i32 = 0;
        while i < n {
        print!("Iteracio {}\n", i);
            i += 1;
        }
    }
}

fn add_one(mut n: i32) -> i32 {
    n += 1;
    return n;
}

fn main() {
    let mut n: i32 = 10;
    print_n(n);
    print!("\nPausa\n\n");
    print_n(add_one(n));
}
```

Listing 3: Final code in Rust, result of transcompiling the initial source code.

3.3. Details of the code generation from the previous example

As it has been stated in the previous section, most of the workload in this project was in the code generation block.
To explain further how it works, we shall use the last phase from the previous example and do it step by step.

The abstract syntax tree resulting from the previous phase has three main nodes, one for the definition of each function (nodes of type `FuncDef`).

Similarly to C, Rust has two main parts when defining a function, the declaration or header and the body or compound. In the declaration it is defined the name of the function, its return type and the necessary parameters, while the body contains the actual implementation of said function.

When traversing the tree, the program visits the nodes starting at the root, `FileAST` and then visiting its children, but a node cannot generate its entire code until all its children nodes have returned. Therefore, we will start at the

last nodes (leafs of the tree, the most indented nodes) and go upwards (to its parents, found in the lines immediately before) in this example. To make it easier, we will do it separately for each function and, more specifically, separately for both the header and body of said functions.

`print_n` function

In the example, when generating the equivalent function `print_n`, it first creates the header and then the body, so we will start with the header too.

```
FuncDef:
  Decl: print_n
    FuncDecl:
      ParamList:
        Decl: n
          TypeDecl: n
            IdentifierType: ['int']
      TypeDecl: print_n
        IdentifierType: ['void']
```

Listing 4: AST for the header of the print_n function

As stated before, the program starts at the `Decl` node when creating the header of the function, but it cannot fully generate said header until all its children nodes have return. Therefore, the first node that will be fully traversed (as in, the program has returned from it) will be the `IdentifierType` for the variable *n*, followed by its parent node `TypeDecl: n`.

- The program visits the `IdentifierType` node, it returns the equivalent type to `'int'`, defined in the file `equivalencies.json` (see section 3.3). Would the file not contain an equivalent type for it, the trancompiler simply would return the same type as in C. In this case, it returns the type `i32` as its the equivalent in Rust.
- Once returned from the `IdentifierType` node, the program visits the `TypeDecl` node, which has the attribute `n` as the name of the variable. Therefore, it returns `n : i32` which indicates that the variable *n* is of type `i32`.
- On the `Decl` node it sees the children that has just returned is of type `TypeDecl`, so it generates a variable declaration: `let mut n : i32` and returns.
- The `ParamList` node, as its name indicates is used to declare parameters for functions' headers. Therefore, the program knows that no variables will be declared in this section so it removes the `let` statement from the expression `let mut n : i32` that was the result of one of its children. Then, it returns the expressions once it has 'cleaned' them.

- The program returns to the `FuncDecl` node, but it still has children to visit, so it visits the other branch, which ends on the node `IdentifierType` that defines the type that the function will return, in this case `'void'`. Therefore it returns nothing.
- When reaching the `TypeDecl` node of the `print_n` function, the program sees its has no type (`void`) so it simply returns the name of the function.
- The `FuncDecl` node is used to represent the headers of a function, so the program will generate the equivalent header in Rust when traversing it. In this case, seeing the type is `void` it will generate: `fn print_n(mut n: i32)` and return.
- Finally, the `Decl` node will see that its children is of type `FuncDecl` so it will not modify the statement and simply return.

At this point, the program has completely generated the header for the `print_n` function, `fn print_n(mut n: i32)`, and is on the `FuncDef` node once again. To complete the function declaration it will now visit the body and generate its equivalent.

```
FuncDef:
  Compound:
    For:
      DeclList:
        Decl: i
          TypeDecl: i
            IdentifierType: ['int']
          Constant: int, 0
      BinaryOp: <
        ID: i
        ID: n
      UnaryOp: p++
        ID: i
      FuncCall:
        ID: printf
        ExprList:
          Constant: string, "Iteracio %d\n"
          ID: i
```

Listing 5: AST for the body of the print_n function

The parent node for the body of a function is always a `Compound` node. When encountering a node of such type, the program simply encases all its child nodes between {} and returns to the parent node. In this case, there is only one child, so the program will visit it, generate the equivalent code, create the block and return. To do so, we will start on the `IdentifierType` node and go from there.

- Like on the header generation, the `IdentifierType` node returns the equivalent to `'int'`, `i32`.

- Once again the `TypeDecl` simply takes its attribute, the variable *i*, and the type from its child node and creates the expression `i : i32`, returning it.
- On the node of type `Decl`, the program encounters a variable declaration, but also a `Constant` as its child, so it visits it. The result of visiting the `Constant` is its value, in this case `0`. Therefore, the program returns to the node `Decl` and creates both the variable declaration and its assignation: `let mut i: i32 = 0;`.
- The parent of `Decl`, `DeclList` always return its children's values.
- The program is again on the `For` node, but to create the equivalent of a *for* statement in Rust, it needs to visit all its children.
- The program first visits the stopping condition, in this case the `BinaryOp` node. When traversing this node, the program simply visits its two children and creates the expression. In the example, the children are `ID` nodes, which always return the name of the variable who's *id* represents, in this case the variables *i* and *n*. Therefore, the program creates the expression `i < n` and returns.
- Now the program encounters the repeating statement, in this case a `UnaryOp`. In the same fashion as just before, the child node returns the name of the variable, *i* and the program generates the equivalent operation. In Rust, the post-increment operation does not exist (neither do the pre-increment or its equivalent with decrement), so the program generates the expression `i += 1`.
- Now the program encounters an extra node, which means this *for* loop only has one statement. This is not possible in Rust, so it will be created the same way as if the loop had multiple statements in it. The node is of type `FuncCall`, which needs to visit its children before proceeding.
- The first child is an `ID` node. Instead of returning the name of the function, this time the program checks if there exists an equivalent function or macro to this function *id*, and returns it instead. In this case, it returns the `print!` macro.
- The `ExprList` contains the arguments for the function call. It will first visit its children node and concatenate them with a comma between them. In this case, it will first find the `Constant` node, which when visited will return the equivalent *string* but changing the formatting expression (`%d`) by the one used in Rust ({}). Then it will visit the `ID` node which, as before, will return the name of the variable. Finally, the program will generate the expression `"Iteracio {}", i` and return.
- The program will be back once more at the `FuncCall` node, but now with all its children visited. Therefore, it will generate the expression with the

name of the function or macro in Rust and its arguments, in this case `print!("Iteracio {}", i);` and return.

- Then, the program returns to the `For` node, but now it has visited all its children already, so it can properly generate the equivalent to the *for* loop. In Rust, *for* loops work exclusively with *iterator*s[12], so it cannot be directly transcompiled to them. Instead, the program will generate an equivalent *while* loop. The way it works is:

  - First creates a block, so that the variables declared inside the loop are not available outside of it.
  - Then appends the declarations found in the `DeclList` at the beginning of the block.
  - The program then generates the *while* loop with the same stopping condition (the result of the `BinaryOp`) as the original loop.
  - It puts all the statements found inside the original *for* loop in the equivalent *while* loop. In this case, the code generated from visiting the `FuncCall` node.
  - Finally, it adds the repeating statement (the result of `UnaryOp`) at the end of the *while* loop and closes both the loop and the block.

  Which results in the following:

```
{
  let mut i: i32 = 0;
  while i < n {
    print!("Iteracio {}\n", i);
    i += 1;
  }
}
```

Listing 6: Result of the code generator visiting the For node and its children.

  And then returns to the `Compound` node.

- The program finally arrives once again at the `Compound` node, which, as stated before, simply surrounds the code resulting from visiting its children with {} and then returns to the previous node.

Once both the header and the body of the function have been generated, the program returns to the `FuncDef` node, where it simply appends them, creating:

```
fn print_n(mut n: i32) {
  {
    let mut i: i32 = 0;
    while i < n {
      print!("Iteracio {}\n", i);
      i += 1;
    }
  }
}
```

Listing 7: Result of the code generator visiting the FuncDef node for the print_n function.

`add_one` function

In the same vein as with the `print_n` function, the transcompiler will first generate the header of the `add_one` function and then the body.

```
FuncDef:
  Decl: add_one
    FuncDecl:
      ParamList:
        Decl: n
          TypeDecl: n
            IdentifierType: ['int']
      TypeDecl: add_one
        IdentifierType: ['int']
```

Listing 8: AST for the header of the add_one function

This part of the tree is extremely similar to the one corresponding to the header of the `print_n` function (see Listing 4), with the only main differences being the names of the functions and the fact that this one returns something.

- The `ParamList` section will be executed exactly the same way it was done in the previous function.
- When visiting the `IdentifierType` corresponding to the return of the function (the one under the `TypeDecl: add_one`) the program will return an actual type to its parent (namely `i32` in this case).
- This time, when the program reaches the `FuncDecl` node it sees that the function has a return type in Rust, therefore in creates the header the same way as before but adding a return type indicator, creating the statement: `fn add_one(mut n: i32) -> i32`.

With this, the header of the function has been created and the transcompiler moves onto the body.

```
FuncDef:
  Compound:
    UnaryOp: p++
      ID: n
    Return:
      ID: n
```

Listing 9: AST for the body of the add_one function

As stated before, the body is a `Compound` node and its children. In this case there are two children of said node, therefore two statements inside the function.

- First it will visit the `UnaryOp` node, which will get the equivalent *id* from its child node, `ID`, and create the equivalent statement. As seen before, the post-increment operation does not exist in Rust, therefore the resulting operation once transcompiled is `n += 1;`.
- When visiting `Return` nodes, the program simply creates a statement in the form of `return + result of its children`. In this case, being its only child the `ID` node, the final result will be `return n;`.
- The program will reach the `Compound` node once more, where again it simply surrounds the result of the children nodes with {} and returns.

Finally, on the `FuncDef` node with all its children visited, the program simply appends the header and the body, creating the transcompiled function:

```
fn add_one(mut n: i32) -> i32 {
        n += 1;
        return n;
}
```

Listing 10: Result of the code generator visiting the FuncDef node for the add_one function.

`main` function

Once again, the transcompiler will first generate the header and then the body.

```
FuncDef:
  Decl: main
    FuncDecl:
      TypeDecl: main
        IdentifierType: ['void']
```

Listing 11: AST for the header of the main function

As it can be seen in the AST corresponding to the header of the `main` function, it differs from the previous two cases in that it does not have a `ParamList` node. That is because in the original source code, no parameters are passed to said function.

- The transcompiler will traverse the nodes that correspond to the type of the return of the `main` function, realising it is void in the same fashion it did on the `print_n` function's case.
- When returning to the `FuncDecl` node, the transcompiler will see that no `ParamList` node is present so it will generate a main function without arguments. The statement it will create is `fn main()`.

Once the function's header has been generated, the transcompiler will move towards the body's nodes.

```
FuncDef:
  Compound:
    Decl: n
      TypeDecl: n
        IdentifierType: ['int']
      Constant: int, 10
    FuncCall:
      ID: print_n
        ExprList:
          ID: n
    FuncCall:
      ID: printf
        ExprList:
          Constant: string, "\nPausa\n\n"
    FuncCall:
      ID: print_n
        ExprList:
          FuncCall:
            ID: add_one
              ExprList:
                ID: n
```

Listing 12: AST for the header of the body function

Again, the body of the function is generated from a single `Compound` node. In this case, four different children spring from said `Compound` node, as there are four statements in the body of the `main` function.

- The first statement is a variable declaration, which starts at the `Decl` node, and will develop in the same way as it has been previously shown in the `print_n` function, resulting in a similar statement: `let mut n: i32 = 0;`
- The second and third statements are function calls that are developed once again like they were in the `print_n` function, the differences being that this time both function calls only have one child to the `ExprList` node, and that the first call (to the `print_n` function) does not have a function equivalent in Rust, as the call is made to another function within the source code.

- Finally, the program encounters two nested function calls (in the AST one is a child to the other). The transcompiler will first generate the `add_one` function call, similarly to the previous cases, and produce the `add_one(n);` statement.
- When the parent `FuncCall` node is reached again by the transcompiler, it will see that the parameters list contains another function call, so it will remove the semicolon from said parameter, remove all traces of indentation and then encase it between parenthesis, creating the final function call: `print_n(add_one(n));`.

Once the program reaches the `FuncDef` node again, it will once more concatenate the header and the body of the `main` function, generating:

```
fn main() {
    let mut n: i32 = 10;
    print_n(n);
    print!("\nPausa\n\n");
    print_n(add_one(n));
}
```

Listing 13: Result of the code generator visiting the FuncDef node for the add_one function.

Once all three `FuncDef` nodes have been visited, the transcompiler will return to the `FileAST` node, where it will concatenate in turn all three functions and, seeing there are no other nodes pending to visit, return the final code in Rust to the main module, where it will be saved to disk.

3.4. Files and customization

This project is mostly written in Python, with JSON files for modifiable settings. Said files can be found in the project folder, `c-to-rs`.

The main files of the project are the Python files and the two JSON files:

- `c-to-rs.py`: Main module of the project. Deals with command line options and file input/output.
- `pre_processor.py`: Preprocessor for the transcompiler.
- `rust_visitor.py`: Code generator for the transcompiler. It can be edited to include specific functions' transcompilation rules.
- `dependencies.json`: User-editable file that contains the dependencies that the Rust project will have.
- `equivalencies.json`: User-editable file that contains the equivalencies between C and Rust, including functions types and compiler directives (`#pragma`s).

Additionally, in the project folder there can be found two other folders: `out` and `examples`. `examples` contain multiple small programs in C that transcompile correctly and were used to develop the main module. On the other hand, `out` is the folder where the transcompiled source code will be generated in, along with the file that describes the Rust project generated to the compiler.

The main customizations a user can do to the project are: changing what C types are transcompiled to, changing what C functions are transcompiled to (used mostly for standard library functions) and changing the dependencies of the final Rust project.

To change what the C types and functions are transcompiled to, the user can change the elements `equivalent_types` and `funtions_with_equivalents` in the `equivalencies.json` file. If a user wants to modify how a specific function is transcompiled (in the project it is set for `malloc`, `calloc` and `abs` functions), they must put the name of the function in the `functions_with_parsing` element in the same file. Then, a specific function must be created in the code generator (`rust_visitor.py` file) with the name `_analize_{function_name}` where function_name is the name in C of the function. This new function will be called at runtime by the transcompiler when encountering the function defined in `functions_with_parsing`.

To add or remove a library to be used on the Rust project (called `crates` in Rust) the user only has to modify the `dependencies.json` file including the name, the version and if it has to be used or not. Additionally, if the `crate` contains a `macro` that is going to be used by the transcompiled version of the code, it is only necessary to indicate it in the `dependencies.json` file and the transcompiler will handle both telling the compiler that macros will be used and that the `crate` contains them.

Finally, the process to generate an equivalent program that uses a function from library is:

- Find a suitable library (`crate`) with said equivalent function[4].
- Add the equivalence in the `equivalencies.json` file.
- Add the `crate` to the `dependencies.json` file with the necessary arguments.
- Transcompile.

All possible customizations include examples already implemented.

[4] All the publicly available crates can be found at *https://crates.io*.

4. Tests and examples

This project was developed similarly to test-driven development. That is, every new feature (in this case, features of the C language to be transcompiled) was introduced via a test (in this project, a source file that included said new feature to transcompile). In short, the iteration for every new feature included in the project was:

- Write a test (source file in C) that includes the new feature to transcompile. If it transcompiles correctly, behaves as expected and there seem to be no problems, this feature is included.
- If it does not transcompile correctly or when transcompiled it does not execute as expected, implement said feature in the transcompiler so it works correctly.
- Test all previous working features, including the new one. If one does not work, refactor until all do.
- Clean up code.[14]

Therefore, all tests under the `examples/` folder of the project should transcompile correctly and produce the same result when executing the source code compiled in C and in Rust.

The main differences between the development of this project and test-driven development were that the tests were not unitary (as in, each test only includes one feature) and that they were not automated.

4.1. What is working

In the current state of the project, most basic functionalities of C are correctly transcompiled to Rust and work in a similar manner. The following can be transcompiled:

- Variable declarations: All variable declarations should work correctly. The only exceptions would be when using a non-existing type in Rust while not providing an equivalent. In this case, the transcompiler will interpret it as the type is called the same way producing expressions in the vein of `let mut n: int64_t;` which would throw a compiler error.
- Function declarations: All function declarations (header and body generation) should work correctly. This does not mean that an error with a statement in the body of the function could exist.
- `struct` declaration and use: There should not be any problems related to the use of `struct`s outside of errors caused from type declarations of its members.

- Function calls: Any type of function calls, including function composition, should be transcompiled properly as long as the functions that are being called are present in the code as well.
- Static array declaration and use: any array allocated at compile time will be translated to Rust correctly, as well as any access to a valid position in it. There might be some errors when using non-integer types to reference an array index, because Rust requires any index to be of type `usize`, which may not be cast properly.
- Loops: Any type of loop (`while`, `for` or `do/while`) should work without issues. Both other types of flow control within loops (namely `contine` and `break` statements) should also work as intended.
- Heap allocation: Most dynamic allocation of objects onto the heap (via `malloc()` or similar) should work correctly. Because of the way Rust operates, those objects will be freed once they go out of scope (similar to smart pointers in modern C++).
- Selection statements: Typical selection statements (`if/else` and `switch/case`, as well as the ternary operator) should work correctly under most circumstances.
- Standard unary operators: All the unary operators with the exception of address of (`&` operator) should transcompile correctly. It is important to note that the address of operator will transcompile to the "borrowing" operator in Rust[13]. Also worth noting that the dereferencing operator (`*`) will not always behave correctly as it requires an `unsafe` block to be used in Rust with raw pointers. Finally, the pre-increment and post-increment operators (`++`) will both transcompile to an equivalent of the pre-increment, as is the case with pre-decrement and post-decrement (`--`).
- Binary operators: All binary operators are transcompiled correctly, including, but not restricted to, bitwise operators. Note that the result of a comparison binary operator, *e.g.*, `<, >, ==` return a `bool` type in Rust instead of 1 or 0.
- `typedef`s: Definition of new aliases for existing types or definition of new types works correctly.
- Explicit casting: Casting a variable of one type into another will work as intended when transcompiling.
- K&R function declaration: This style of function declaration, while mostly obsolete, is also supported because the `pycparser` library also supported it and was non-issue to create that special case.

4.2. Limitations

While most basic programs will transcompile correctly to Rust using this project, there are still some limitations regarding, but not limited to:

- Global variables: While constants work as is in Rust, mutable global variables in Rust are not safe (because when two threads access the same variable it is considered undefined behaviour, therefore `unsafe`). The workaround created by the project when encountering is surrounding any global variable usage with an unsafe block which, while can work in most single-threaded programs, is not the correct way to do it.

- `char` usage: In C there are two main uses for variables of type `char`, both as a character variable or as a short integer. Both of uses are available in Rust, but there is a clear distinction between a numeric variable (types `i8` and `u8` in Rust) and character variables (type `char`). Therefore, any program that makes use of `char`s in both ways will not work as intended when transcompiled as it is programmed to all variables of a type to transcompile to only one type in Rust.

- `char` as a string: There is a `str` primitive type in Rust, therefore most standard library methods that are related to text are expected to use this type instead as of an array of `char`s like in C. Depending on what usage of text there is in the program to be transcompiled the final code in Rust may or may not work.

- Returning an operation: When returning a numeric type in C, if the return statement contains an operation (like pre-subtraction) the Rust compiler will throw an error as the proper return type for such function would be `()`, the operation type.

- Pointer usage: Only very basic pointer usage is supported (only when allocating in the heap via allocator functions). Creating raw pointers from existing variables is not supported and will not transcompile correctly. Similarly, changing the types of a pointer, *e.g.*, reading a floating point binary codification as if it were an integer, will not work even if there exists an equivalent in Rust (it is unsafe to). Moreover, dereferencing anything that has not been allocated in the heap via traditional ways will not transcompile because dereferencing a pointer to a non-compiler controlled memory location is undefined behaviour. Additionally, when allocating anything on the heap it will initialize those memory positions to 0 (equivalent to `calloc` in C) regardless if the original program does not, as not initializing memory that could be referenced to would also cause undefined behaviour.

- Printing non-standard types: While the basic printing functions are transcompiled correctly, some types may not be displayed properly and thus throw a compiling error. As an example, trying to print an `enum` will throw a formatting error.

- Variable naming: There might be some issues when naming a variable with the same name as a member of an `enum`, as the transcompiler will not know if it is a new variable or an actual reference to the `enum`.
- Dependency on undefined behaviour: Anything that depends on undefined behaviour (like overflowing an integer or leave memory initialized without freeing it) may not work as intended.
- Unions: The C-style `union` is considered unsafe in Rust, therefore it was not included. This was changed once this project was completed, and now it is safe to initialize unions and to read the active fields of a union.
- Different type operations: When doing an operation between two different types of variable (like adding a float and an integer), the Rust compiler will throw an error for type mismatch unless there is an explicit cast for one of the variables.
- `switch/case` must have a default: In Rust, the equivalent to `switch/case` must always contain a default statement or else it will throw a compiler error.

4.3. Examples

In this section, several examples of C code and its transcompiled Rust versions will be displayed, as well as a brief commentary on them.

Example 1

```
#include <stdio.h>
enum WebEvent {PageLoad, PageUnload, Test};

void inspect(enum WebEvent event)
{
  switch(event)
  {
    case PageLoad:
      printf("page loaded\n");
      break;
    case PageUnload:
      printf("page unloaded\n");
      break;
    default:
      printf("not recognised\n");
      break;
  }
}

void main(void)
{
  enum WebEvent load = PageLoad;
  enum WebEvent unload = PageUnload;
  enum WebEvent test;

  test = Test;

  inspect(load);
  inspect(unload);
  inspect(test);
}
```

Listing 14: Small program to test enums and switch/case statements.

```
enum WebEvent { PageLoad, PageUnload, Test }
fn inspect(mut event: WebEvent) {
  match (event) {
    WebEvent::PageLoad => {
      print!("page loaded\n");
    }
    WebEvent::PageUnload => {
      print!("page unloaded\n");
    }
    _ => {
      print!("not recognised\n");
    }
  }
}

fn main() {
  let mut load: WebEvent = WebEvent::PageLoad;
  let mut unload: WebEvent = WebEvent::PageUnload;
  let mut test: WebEvent;
  test = WebEvent::Test;
  inspect(load);
  inspect(unload);
  inspect(test);
}
```

Listing 15: Transcompiled version of the previous program.

As it can be seen, the `enum` declaration is practically identical both in Rust and C, with the exception of the last semicolon.

The `switch/case` statement in C has the `match` equivalent in Rust, which does not require a `break` for each case as well as having the `default` keyword changed to a _.

Finally, every time an `enum` is used, in Rust it is required to state the original `enum` type. In this case, it can be seen that `WebEvent` is present anywhere where an `enum` of such type is used.

Both programs output the same when executed.

Example 2

```
#include <stdio.h>
#include <stdlib.h>

struct vessel
{
  float first_var;
  int second_var;
};

void main(void)
{
  struct vessel ves = { 2.0, 4 };
  int *arr = malloc(sizeof(int)*4);
  int index = 0;
  do
  {
    arr[index] = ves.second_var;
    ves.second_var--;
    index++;
    printf("Second struct var is: %d\n", ves.second_var);
    printf("Last array assignement was: %d \n", arr[index-1]);
  } while ((int)ves.first_var < ves.second_var);
  free(arr);
}
```

Listing 16: Small program to test struct and memory allocation.

```
struct vessel {
  first_var: f32,
  second_var: i32,
}

fn main() {
  let mut ves: vessel = vessel {first_var : 2.0, second_var : 4};
  let mut arr: Box<[i32]> = Box::new([0i32; 4]);
  let mut index: i32 = 0;
  loop {
    arr[(index) as usize] = ves.second_var;
    ves.second_var -= 1;
    index += 1;
    print!("Second struct var is: {}\n", ves.second_var);
    print!("Last array assignement was: {} \n", arr[(index - 1) as usize]);
    if !((ves.first_var as i32) < ves.second_var) {break}
  }
}
```

Listing 17: Transcompiled version of the previous program.

This example shows various things:

- `struct` declaration is very similar in both C and Rust.
- When creating the array in the main function, Rust requires the code to have an initialization of the memory to allocate, contrary to C's `malloc` function, which does not initialize. In a sense, Rust does not have a `malloc` equivalent but a `calloc` one.
- The `do/while` loop is translated into a `loop`, which is an endless loop that will only exit through a `break`. In this case, to generate the equivalent it will do the same evaluation as the `while` in C but negated, only breaking out of the loop when it is no longer true (same behaviour as C's `while`).
- In the example, when initializing the `struct` the first variable is declared as `2.0`. This has to be explicitly done this way, as the Rust compiler would throw an error if initializing a floating point variable as an integer.
- There is a cast when comparing the first and the second variable of the `struct`. In Rust, comparing two variables of different nature (integer and floating point, in this case) does not have a defined outcome. Only comparing two types of the same nature will work, if not the compiler will throw an error.
- There is no `free` equivalent in Rust. That is because when any variable, be it allocated statically or dynamically, is freed when going out of scope. In this case, it will go out of scope when the block it is generated ends, *i.e.*, when the `main` function exits.

Example 3

```
#include <stdio.h>
typedef int integer;
void main(void)
{
  integer i;
  integer a = 1;
  for (i = 0; i < 15; i++)
  {
    if (i <= 3) continue;
    else if (i >= 9) break;
    else {
      a *= i;
      a = !(a%2) ? a/2 : a;
    }
    printf("Current iteration: %d\n", i);
  }
  printf("Final result: %d\n", a);
}
```

Listing 18: Small program to test breaks and conditionals.

```
type integer = i32;
fn main() {
  let mut i: integer;
  let mut a: integer = 1;
  {
    i = 0;
    while i < 15 {
      if i <= 3  {
        i += 1;
        continue;
      }
      else if i >= 9  {
        break;
      }
      else {
        a *= i;
        a = if (!a % 2) != 0 {a / 2} else {a};
      }
      print!("Current iteration: {}\n", i);

      i += 1;
    }
  }
  print!("Final result: {}\n", a);
}
```

Listing 19: Transcompiled version of the previous program.

This last example shows the transcompilation of the `typedef` statement, in this case the `type` statement in Rust which has the exact same behaviour.

It also shows once again the `for` loop transcompilation to a `while` loop, but also shows other flow control options. The first that can be found is the `continue` statement. In this case, being a while loop, the repeating statement has to be executed once before continuing to the next iteration, or else it could get stuck in some cases. Later there is the `break` statement, which works in the same way it does in C.

There is also the transcompilation of a ternary operator. Rust does not directly have this operator, but it does have an equivalent with the `if/else` statements, which is used here.

For more examples, see the `examples` folder inside the project folder.

5. Conclusions

The objective of this project was to design and implement a transcompiler from C to Rust. With this project, most of what it set to do was achieved. It is fully functional when working with the features previously discussed while being relatively easy to use and extend or customize.

The result of this project provides the user enough tools to transcompile basic C programs that do not make use of the language's more advanced features in an easy way, creating an equivalent program in Rust that, in theory, has the same behaviour as its C counterpart but enforcing better practices in regards of memory and variable ownership through Rust's own compiler.

While most programs may not transcompile at first because library usage is almost always present, the transcompiler offers enough customization options that a user can get an equivalent program in Rust without much trouble, just changing the settings in the customization files.

Finally, with this project it was seen why the old compiler design philosophies still apply and that is why they are still used today, and that it is much easier to create a compiler when using said designs as well as tools proven to work.

5.1. Future work

While this project is completely functional in its set of features, there is still room for improvement in the form of new features (like concurrency support, pointer arithmetic or pointer transmutation) and expanding on the ones it already has (better support for memory allocators, improvements in reference creation and support).

Furthermore, if down the line it could seem feasible and of any utility, a custom parsing method could be created to encompass all phases of a compiler in the implementation of this project.

It has been stated before that there is an absence of an intermediate phase between the abstract syntax tree generation and the code generation. In this case it worked out just fine traversing the AST and generating the code on the go because C programs' syntax and structure is relatively similar to the one found in Rust. However, if the source language was not the of the C-style family of syntax, like Prolog or Haskell (both of which have syntaxes very different to that of C), then the use of translation tables would be pretty much mandatory to prevent huge amounts of backtracking when traversing the tree, so it could be a good idea to redo this part if there was any interest to make this project compile more than one language to Rust.

Another reason to create said tables would be that it would allow for a more idiomatic Rust result when transcompiling, therefore increasing the correctness of the result and most likely its performance.

All in all it is a complete project, but with the correct intent much can be done to further expand it.

References

[1] Metcalf, M. (2011). The seven ages of Fortran.

[2] "FAQ - The Rust Project". Retrieved from https://www.rust-lang.org/en-US/faq.html, 2017-12-05.

[3] The Rustonomicon. Retrieved from https://doc.rust-lang.org/beta/nomicon/README.html, 2018-01-15.

[4] Mogensen, T. Æ. (2000). Basics of Compiler Design, Kursusbog for Datalogi 1E, Vol.5, 2. edition. Datalogisk Institut, Københavns Universitet.

[5] Ilyushin, E. and Namiot, D. (2016). On source-to-source compilers. Vol.4 .

[6] Eli Bendersky. "pycparser". Retrieved from https://github.com/eliben/pycparser, 2017-12-09

[7] David Beazley. "PLY (Python Lex-Yacc)". Retrieved from http://www.dabeaz.com/ply/, 2017-12-09

[8] Emscripten. Retrieved from https://kripken.github.io/emscripten-site/, 2018-01-20.

[9] BaCon - BASIC to C converter. Retrieved from http://www.basic-converter.org/, 2018-01-20.

[10] CoffeeScript. Retrieved from http://coffeescript.org/, 2017-12-15.

[11] TypeScript. Retrieved from https://www.typescriptlang.org/, 2017-12-15.

[12] The Rust Programming Language. Retrieved from https://doc.rust-lang.org/1.2.0/book/, 2018-01-18.

[13] The Rust Programming Language. 2. edition. Retrieved from https://doc.rust-lang.org/book/second-edition/, 2018-01-15.

[14] Beck, K. (2003). Test-driven Development: By Example. Addison-Wesley Professional.