Extending the Alpine Compiler

Project Report

Alice Potter, Cassio Manuguerra, Jules Schneuwly

1 Introduction

During the "Computer Language Programming" course, we had the opportunity to build a compiler for a newly designed programming language called Alpine. Our journey began with implementing an interpreter, which serves as the final step in the pipeline and is responsible for executing the Typed Program it receives.We then proceeded to develop the Parser, which takes a sequence of tokens provided by the Lexer and generates an abstract syntax tree (AST). This AST represents the hierarchical structure of the source code, making it easier to analyze and transform. Following the Parser, we implemented the Type Checker. The Type Checker ensures that the program is well-typed by verifying that the types of expressions are consistent with the types of variables, functions, and constants. It checks that functions are called with the correct number and type of arguments and that variable usage aligns with their declared types. Additionally, the Type Checker infers types for expressions that lack explicit type annotations, computing and storing these types in the AST. It also performs name resolution, ensuring that every variable and function call is linked to a unique definition. Overall, the labs provided us with a comprehensive understanding of compiler construction, from lexical analysis to type checking and execution.

For this project we decided to take our compiler a step further by adding two extensions :

1. **Firstly, we aimed to use C as a compiler target.** This extension involved creating a transpiler similar to the one we had previously written for Alpine to Scala. The challenge was to handle the lower level of abstraction in C while ensuring the compilation of pattern matching constructs and polymorphic containers. This extension not only made our compiler more versatile but also provided valuable experience in addressing interoperability and complexity in language features.

2. **Secondly, we implemented support for Open Unions.** In a previous lab, we developed code generation for a subset of Alpine that excluded polymorphism, meaning each expression had to be assigned a concrete type at compile-time. Our goal with this extension was to lift this restriction to support open unions, allowing more flexible and dynamic type handling. To achieve this, we needed to represent an existential container in the target language. This container can hold arbitrary data along with the necessary type information, enabling the program to understand what the data represents at runtime. By implementing this extension, we allowed the program to match and process the contents of these containers dynamically.

In this report, we're going to share how we brought these two extensions to life. We'll talk about the challenges we faced and the choices we made. So, let's dive into the details and see what we've accomplished.

2 Transpiler to C that supports Open Unions

2.1 Set-up

In the first step of extending our Alpine compiler to target C as the output language, we created a file named `CPrinter.scala` within the Codegen directory. This initial task was inspired by the transpiler we developed from Alpine to Scala in Lab 05. To maintain consistency and leverage our existing knowledge, we structured the `CPrinter.scala` file similarly to the `ScalaPrinter.scala` file implemented in Lab 05. e reused the concept of the `Context` class to organize the output and structure of the C code. The output, meaning the

C file, is returned in the form of a string.

The main challenge we faced was determining how to structure the C file. While Alpine and Scala have similarities, C is structured in a very different manner. To ensure that the transpiled file compiles correctly, we created five different string builders to store the output. These string builders are:

1. `Context.initialOutput` used for the declarations at the top of the C class (type declarations, and basic function declarations and implementations),
2. `Context.globalVariables` handles global variables declarations.,
3. `Context.initializeGlobals` used for global variables initialization. It needs to be inside a method because in C, you cannot call the memory allocation functions outside of a method.
4. `Context.functionOutput` used to declare functions and their body above the main function,
5. `Context.output` used to store the body of the main function.

The global C output is a concatenation of these string builders. This structured approach ensures that our transpiled C file is well-organized and compiles successfully, adhering to the requirements and constraints of the C programming language.

2.2 Transpiling the basic types

The first transpiling task we tackled was converting the basic types to C. We chose to represent each type as a C structure featuring two key attributes :

1. a `payload` attribute that holds the variable's actual value,
2. a `type` attribute that identifies the corresponding type of the value.

This decision was strategically made because C does not natively support pattern matching, a feature integral to Alpine. By structuring each basic type in this way, we create the necessary infrastructure to implement a version of Alpine's pattern matching in C. Therefore, in every transiplation from Alpine to C, we include the following code at the beginging of the class :

```
typedef struct {void* payload; TypeEnum type;} Int;
typedef struct {void* payload; TypeEnum type;} Float;
typedef struct {void* payload; TypeEnum type;} Boolean;
typedef struct {void* payload; TypeEnum type;} String;
```

Notice the use of the `TypeEnum` structure. This structure was created to allows us to refer to each type by its name while storing them as integer. The code of the structure `TypeEnum` is included before the definition of the the varaible structures :

```
typedef enum {
    IntType = 0,
    FloatType = 1,
    BooleanType = 2,
    StringType = 3,
    RecordType = 4,
    AnyType = 5
} TypeEnum;
```

The declaration of the structure that represent the variables as well as the `TypeEnum` strucutre are both added to the `Context.initialOutput`, they are displayed at the top of the outputted C code.

2.3 Transpiling the Records

In the process of transpiling Alpine to C, implementing records required a thoughtful approach to accommodate the complex data structures common in high-level programming languages. Like the basic types, we represented the record type in Alpine as a structured type in C. This structure includes several attributes to manage the intricacies of record handling and support the implementation of pattern matching, which is not natively supported in C.

The C structure for records includes the following attributes:

- `nb_fields`: A count of how many fields the record contains,
- `fields`: A pointer to an array of pointers, each pointing to a field within the record,
- `type`: An attribute that identifies the record type.
- `name`: A character pointer to a unique name that identifies the record type. This name is crucial for distinguishing between different record types when implementing pattern matching in C.

```
typedef struct {size_t nb_fields; void** fields; TypeEnum type; char* name;} Record;
```

Let's consider how to assign a unique name to the record structures, taking into account their identifiers and fields. Our goal is to uniquely identify structures that may have the same identifier but are differentiated based on their fields. Importantly, the order of fields within the record doesn't influence its identity. For example, the following records are considered the same type despite the order of their fields:

- `let x = #person(name: "Bob", age: 18)`
- `let y = #person(age: 21, name: "Charlie")`

To construct a unique name for such records, we start with the record's identifier (for example, "person"). Next, we iterate over the fields of the record, focusing on the types of these fields. To ensure consistency regardless of field order, we sort these types alphabetically before adding them to the name. Therefore, the unique name for the records mentioned would be "person_I_S" (where 'I' stands for Integer and 'S' for String). We focus solely on the types of the fields, not their identifiers, because identifiers are not mandatory for determining the type of a record. This means that even the following record would be considered the same type as the others, as long as the types of the fields match, regardless of how the fields are labeled or ordered.

```
let z = #person("Nina", 22)
```

When a field within a record is itself a record, we use its unique name to represent its type in the parent record's unique identifier. This recursive approach helps maintain clarity and uniqueness even in nested structures, ensuring that each record's identity is accurately reflected in its name.

To create and initialize a record in our Alpine-to-C transpiler, we start by iterating over the record's fields from the Alpine code. For each field, we convert it into the corresponding C type and gather these converted fields into a C list.

After compiling the list of fields, we proceed to actually create the record using a `createRecord` function that we've defined below the structure type definitions in our code. This function is designed to construct a new record based on the `Record` structure type we previously outlined. We were particularly cautious about memory management during this process; we used `malloc` to allocate memory for each new record to ensure proper memory management at runtime.

However, a significant challenge we've encountered is managing the memory deallocation for these dynamically created records. As of now, we haven't developed a mechanism to determine the optimal time within the code to free the memory we allocated. This is a complex aspect of memory management in C, especially

when dealing with life cycles of data structures that aren't inherently managed by the language. We plan to further investigate and implement a strategy to handle memory freeing appropriately to avoid memory leaks in our transpiled code.

2.4 Bindings, Functions, Applications

The implementations of the visit function, binding, and application were inspired by the work we did in Lab 05. In this lab, we adapted our existing knowledge and techniques to the C programming language. We meticulously adjusted the syntax and semantics to suit C, ensuring a seamless transition from the previous Scala implementation. To generate the output, we utilized our various string builders, which allowed us to efficiently construct and organize the resulting C code.

2.5 Print method

To implement the `print` method in our Alpine-to-C transpiler, we first needed to determine the type of the variable whose value we intended to print. This type-checking is essential because, in C, the `printf` function requires that the type of the variable being printed be specified to format the output correctly.

Once we identified the type of the variable, we then called the appropriate `printf` method tailored to handle that specific type.

In cases where the variable's type is a record, the process becomes slightly more complex. We iterated over the fields of the record, checking the type of each field in turn. For each field, we again called the appropriate `printf` method based on its type.

2.6 The Any type

We represented an Any type similarly to basic types by defining a structucture that contains both the payload and the actual type of the Any.

```
typedef struct {void* payload; TypeEnum type;} Any;
```

Our code is built on encapsulating an internal type within every possible value. Because we cannot represent a type in C that has no defined type, we just gave to the any its actual type but let expressions match on it as if it had no type. This approach allows us to easily incorporate the Any type with the rest of the code. For instance, the expression `let x: Any = 2` followed by `let y = 5 + x` will work fine since the internal type of `x` is `IntType`, and it will be cast accordingly. Our existing match functionality already supports this, as it checks against the internal type. One of the challenges we faced was ensuring that the `visitAscribed` function works seamlessly with the Any type.

2.7 Ascriptions

When representing ascriptions in C, the initial challenge was to manage variables created within the ascription. In the example below, this would be the integer 2.

```
let x = 2 @ Any
```

We decided to create temporary variables, declared and initialized on the spot. These variables, named `temp_<unique number>`, hold the value of the left part of the ascription, the `inner` expression.

The `Context.globalVariables` holds the declarations of other variables that store the results of type casting. We created one variable for each type. `Context.globalVariables` also holds a variable for the final result of the ascription:

```
Any ascribed_inner;
Boolean ascribed_inner_b;
Int ascribed_inner_i;
Float ascribed_inner_f;
String ascribed_inner_s;
```

All three type of ascriptions explained below are all done in the `initializeGlobals()` functions, this way the result of the ascriptions can be used inside match cases, function parameters etc...

2.7.1 Widen (@)

Widening involves casting a variable to type `Any`. We implemented a function at the top of the C file to handle the casting: `Any* typeToAny(void* payload, TypeEnum type)`.

The result of the function call is stored in `ascribed_inner`, particularly if the Widen ascription is within another ascription, such as `let x = ((2 @ Any) @! Int)`. If it stands alone, the result is stored in the transpiled variable `x`.

This approach to transpile Ascriptions allows for handling recursive ascription declarations, a key feature of the Alpine Language.

2.7.2 Narrow Unconditionally (@!)

Narrowing unconditionally is similar to Widening. We created functions for typecasting based on the specific type:

```
Int* anyToInt(void* payload);
Float* anyToFloat(void* payload);
Boolean* anyToBool(void* payload);
String* anyToString(void* payload);
```

The difference lies in handling recursive definitions of ascription. We needed to track which type was inside the `Any`, which required using the `ascribed_inner_<initial of the type>` variables and manually changing them depending on the type of casting being performed.

2.7.3 Narrow Conditionally (@?)

Narrowing conditionally was tougher to implement. We had to implement checks through if conditions to verify if the casting was possible, if we tried casting an Any holding a different type than the one on the right side of the ascription, we should return `#none`, else return `#some(<variable>)`.

We then created the record structures to represent the `#none` and `#some(<variable>)`.

2.8 Pattern matching

When implementing pattern matching in C for our Alpine compiler, we initially considered using a traditional approach of a series of 'if-else' conditions to handle the various match cases. This method would sequentially check each pattern against the given value and execute the corresponding code block upon a match. However, after some deliberation, we opted for a more compact and elegant approach.

We decided to employ the ternary operator in C, a conditional expression that takes the form:
`condition ? success_case : failure_case`. This allowed us to express our pattern matching logic in a more concise manner.

To facilitate this implementation, we created a helper method within the `visitMatch` method of our compiler. This helper method orchestrates the recursive structure of the ternary operations. Each call to the

method handles a part of the pattern matching, checking a condition, and either resolving to the corresponding success case or recursively proceeding to the next possible match.

This helper method would first check if we were in a function, in a binding or in the main (which we could have made way more easy by just creating a match function that we would call with its identifier, but we did not think of it soon enough), and then for each pattern evaluate its content via a match to check whether it is a type match (Binding) or a value match (ValuePattern or RecordPattern). The fact that we did not think of implementing pattern match with a function forced us to devise a way of handling both the cases where expressions needed to be evaluated directly in the match scrutinee, or if they were previously evaluated in a binding.

This led to several edge cases, as instance for records matching where we needed to create a temporary record in the global variables so that we could correctly check its values/types.
Another edge case were the parenthesized expressions with an inner record, as our helper function was built in a way were it would not recursively check the expression in parentheses, and because the parenthesized expressions are seen as a ValuePattern and so we cannot just remove the parentheses and continue because we are in a wrong match case. We did not have time to handle the case, but an easy (and ugly) way to do it could be to add a check if the expression is a parenthesized expression handle rewrite all the helper function content in the condition to handle everything from here again.

3 Conclusion

This task was quite challenging due to the low-level nature of the C programming language. Despite this, we successfully constructed a C transpiler that translates our higher-level code into C code. One aspect we did not have time to implement was memory deallocation. However, we considered a solution involving the creation of a list within the context to hold all the variables that have allocated memory. At the end of the program, just before the `return 0` statement in the `main` function, we would iterate over this list and free all those variables. This approach would ensure proper memory management and prevent memory leaks.