Steps towards a verified compiler for a subset of Ada

Fanny Canivet, under the supervision of Magnus Myreen
Chalmers University of Technology, Gothenburg, Sweden

March – July 2020

Contents

Note about the location of the internship

This internship began with 2 normal weeks at the office in Chalmers and has been led from home after that, with 3 months in total spent in Sweden and the remaining time back in France.
I had contact with my supervisor and an other member of the team, Hira Syeda around once a week, and sometimes chats and mails for questions outside the video meetings.

1 Introduction

I have been doing a 5 month long internship under the supervision of Magnus Myreen, and worked along with Hira Syeda on the subject of a verified compiler for a specific language. The chosen language was SPARK, a subset of Ada. This language is used in industry in various fields that need high safety standards like rail transport, for example, because it has the particularity that programs can be formally proved to be correct, so it ensures a quite high reliability.

Nevertheless, even if the SPARK code is proved to be correct, errors can be created at the compilation step, and then a program that was considered correct could exhibit faults at execution time. By having a formally proved compiler, one can ensure that such issues cannot happen and have an even greater confidence in programs. The properties proved from the source code would then still be correct in the binary produced.

The goal was then to write such a compiler for the SPARK language, using the HOL4 theorem prover, and the subject of the internship was to compile to the Pancake language, which is part of the CakeML compiler and thus already has a complete and formally proved compiler.

I wasn't able to finish this task during the internship, as I have been at the beginning of the project, I will then present what I managed to do.

First I will present general information about the subject and what had been already done, then I will present the formalisation of the SPARK language and the first steps of compilation that I began before the end of the internship: the flattening process and the parsing (as well as lexing). This report finishes with a summary of what works remains to be done to complete this project.

2 Context

2.1 Presentation of SPARK

SPARK is a formally defined language that is a subset of Ada. It aims at being as complete as possible and to remain unambiguous so programs can be proven. SPARK is used in several safety-critical system, as well as security related programs: railway safety [3], hospital information systems [4] and even communications on the ISS [2]. SPARK is therefore not only a tool for researchers, but a language with practical applications, which motivates its choice for this project.

GNATprove, the SPARK compiler, comes with an integrated prover: all programs in SPARK have to meet certain requirements to pass the compilation step. The programmer can even add its own conditions (for example assert that the result of a function will not exceed a certain number) that the compiler will add to its requirements.

Here are some of the requirements of SPARK, that the Ada language does not have:

- There are no access types (pointers). This requirement makes it easier to prove things, because it can be tricky to track how memory is modified when the way it is accessed is itself a variable.
- No side effects in expressions. This ensure the flow of computations is under control, making it easier to prove certain properties. This is a limitation only for functions, and it will be seen in subsection 3.3
- No aliasing. This means that a location in memory can only be referred to by a single variable. For example, if a function `f` takes parameters `a` and `b`, you cannot call the function with the

same variable (`f(x,x)`). This prevents bad things from happening, for example when copying back the value for `x`, you would not know if it gets the value of `a` or `b`. This will be explained with more details in subsection 3.2

The AST of the SPARK subset that I have been working on is available in appendix A of this document. This AST, as well as the parsing rules also available in the appendix B have been made using existing work of formalisation, but also the reference manuals for Ada [5] and for SPARK [6].

2.2 Pancake

The SPARK compiler that I worked on will target Pancake[1], which is a Pascal-like language that is developed jointly by the Division of Formal Methods[2] at Chalmers and the Model-based Computing Systems group at KTH University, Stockholm.

Pancake is an imperative language that includes among others features loops, structs, function calls and foreign functions interface. This language would compile to the lower part of the CakeML compiler, which is a verified implementation of an ML like language. So the compilation of Pancake programs ensure correct generation of machine code, and having a verified compiler from SPARK to Pancake will keep this property of correctness.

This language is still being developed, and if the compilation passes to the CakeML base has been proven correct, there is still a need in updating the CakeML structure to support the Pancake compilation.

2.3 Background works

As I began to work at the very beginning of the SPARK to Pancake compiler project, there were no list of "*similar works*" I could use as a basis for mine. Thus, I started by making such a list that would make writing the compiler easier.

It appeared that a formalisation of a subset of SPARK [12] had already been done in Coq. The goal of this formalisation was to prove that GNATprove – the prover included in the SPARK compiler – was correct. I only looked at the formalisation work, as the further work done there was not linked with mine.

All the source code can be found in their GitHub repository [8].
I then rewrote the formalisation in the *lem* language based on this work, I will present with more details what I actually did in 3.1

2.4 Tools used

During my internship, I used several specific tools that I did not know about. I will briefly present them and explain how they were helpful for this project.

lem *lem* [9][10] is a language that has a very close syntax from the OCaml one. It has been designed to write semantic definitions and can code in a variety of different languages like Coq, HOL4, HOL/Isabelle, etc. I used it to write the formalisation of SPARK and for the flattening step for compilation. It has been very useful for producing HOL4 definitions but still being able to test easily the functions by producing OCaml code.

HOL4 HOL4 [7] is an interactive theorem prover for higher order logic. It has been used to verify all the CakeML compiler. I used it for writing the parser and the lexer of SPARK, but I haven't led much proofs with it, and mostly defined functions. The files written in *lem* also generated HOL4 code, that will be used later in the project to have the verification done.

[1] Since Pancake is a very young project, I could not find any publicly available documents.
[2] `https://chalmersformalmethods.github.io`

CakeML CakeML [1] is a ML-like language that has the particularity to have a complete formally verified compiler. The compiler that was started in this internship will produce Pancake code, that will be compiled by the CakeML compiler. The CakeML compiler source code has been written in *lem* and HOL4, so it has been very useful for me, so I could see it and take example of what has been done for the SPARK compiler. I particularly found it helpful for the parser and the lexer, because most of general functions and features could be immediately adapted in my case.

3 Formalisation of SPARK

3.1 Base of the formalisation

To write the formalisation of SPARK, or at least a large subset of SPARK, I worked on an existing one, presented above. Nevertheless I changed a few things, that I will present and explain in this section.

3.1.1 Structure

In the existing formalisation they were two different data structure to store the existing variables, procedures and types declared in the program. The *state* that stores the values of variables in the scope and the *symboltable* that contains all objects declared in the program and their definition (type for variable, parameter specification and body for procedures, for example), and everything had been renamed in a previous step to remove name clashes (all objects were referred by a unique number) and to fill the symboltable.

I decided to do this differently, and to have only one structure combining the information from the symboltable and the state. This allowed me to evaluate the program without renaming all objects and without any preprocess.

The resulting symboltable includes then all objects, their definition and their value if any (for variables only) that had been declared so far. It is organised using the different frames. A frame represent a stack frame and includes the actual level (syntactic nesting level of the procedure or function declaration) and all objects declared in this level. The symboltable is then just a list of those frames. This structure allows to easily remove the stack frames that will not be useful in a procedure or function call, so it can only refer to the variable it is supposed to.

The information stored for each objects are the following:

- variable: name, value (uninitialised if no value given), mode (tells if the variable can be written or not), alias (for aliasing prevention, see 3.2)
- procedure and functions: name, level where it is declared, its declaration
- type: name and its declaration

I also added in the symboltable a *clock* field, that takes a natural number that is decreased at each evaluation step that could loop (call to a function or procedure or in a loop step) to have a terminating evaluation even if the program does not terminate.

This is part of the evaluation methods I used, the functional big-step semantics [11], that has been used for the CakeML formalisation as well. This way of writing semantics allows to write in a functional style and is easy to read, and the use of a clock enable the handling of non terminating programs in the semantics.

Here is a small example of how the symboltable behaves:

```
procedure main is
  x : Integer;
  y : Boolean := true;

  procedure plus1(a : in out Integer) is begin
    a := a + 1;
```

```
  end plus1;
begin
  x := 0;
  plus1(x);
end main;
```

At the beginning of the statements part in the `main` procedure (after the `begin` keyword), the symboltable would contain the information of the variables `x` and `y`, with `x` being uninitialised, and the definition of the procedure `plus1`. The first statement will update the symboltable, as the variable `x`, is going to have now the value `0`. The second statement is a call to a procedure. There a new frame is created, that will have a greater level than the previous, that will contains the variable `a`, that will be initialised with the value of the passed parameter `x`. This new frame will vanish after the call.

3.1.2 Features included

The existing formalisation included the following features:

- Type declaration: subtypes, arrays and records are allowed
- Variable declaration
- Expression: all classical expressions except function calls
- Statement: procedure calls, if, while and assignment

Most of these features are quite straightforward to evaluate and has been done following the existing formalisation, once intermediate functions to interact with the new standard of symboltable has been done.

For the while and call statement, I just added a clock check (if it lowers to 0, it raises an error) and it decreases each time this kind of statement is evaluated.

I then added some features that were missing in the existing formalisation, and some checks for the code that were not supported. The list below includes the check to detect aliasing and a type check, and the explanation for the adding of functions and other type of loops than while.

3.2 Aliasing prevention

3.2.1 Definition

In SPARK, aliasing is forbidden, that is to say, it is forbidden to have several variable names referring to the same object. The goal is then to find in which cases it can appear and how to detect it.

In the body of a function, when defining a variable with the help of another one, it doesn't create aliasing because the value is copied, even with potentially large values like arrays (eg. `let x := y` doesn't create aliasing, but `x` is initialised with the same value than `y`).

Aliasing can then occur mostly when calling a procedure or a function, for example with the following procedure:

```
procedure f(x: in out Integer; y: in out Integer) ...
```

There is aliasing if `f` is called with twice the same parameter (`f(a, a)` for example). Aliasing also exists when a global variable `V` is used inside the procedure but is also passed as a parameter, for example:

```
procedure g(x: in out Integer) is begin
  V := 2; x := 3;
end g;
```

In this example, calling `g(V)` is forbidden.

Aliasing only occurs with `out` parameters, and it is forbidden because otherwise the order in which values are copied back at the end of a procedure would change the result (in the first example, if `f(a, a)` is called, one cannot know if `a` gets the value of parameter `x` or `y`).

These examples illustrate the two ways of creating aliasing, I will then explain how I managed to detect these cases to ensure no aliasing in a SPARK program.

3.2.2 Between a parameter and a global variable

This is the case: `f(x)` when `x` is in the scope of `f`

To prevent aliasing errors from this case, what I do when evaluating a SPARK program is creating a tag for every variable in the state that tells if the variable has been passed as a parameter before or not (`AssignForbidden` or `NoRestriction`). Contrary to what I first believed, once a variable has been passed as an `out` parameter, not only assigning it is forbidden but reading it too. So when both fetching the value of a variable or assigning one, the aliasing tag is checked, if its value is `AssignForbidden` then the semantic evaluation function raises an aliasing error.

When a variable is declared, this tag has the value `NoRestriction` and is updated to `AssignForbidden` just after copying the value of the actual parameters to the formal parameter names. After evaluating the body of the procedure, this tag is restored to `NoRestriction` just before copying back the values.

3.2.3 Between two parameters

This is the case `f(x, x)`

To prevent aliasing from this case, it is a bit more complicated because I wanted to be a bit more precise. That is to say if `A` is an array, then `f(A(1), A(2))` is not aliasing because the parts of the array concerned do not collapse, but with the previous method, the entire array `A` is taged as `AssignForbidden` and can't be used anymore. The way to find this kind of aliasing is then to check parameters, comparing their names to the ones checked before, if they are referring to the same object exactly or if one is part of the other (eg `A` and `A(1)`) then it raises an aliasing error, else it just add the name of this parameter to the list of checked parameters. I use this list after testing all parameters to update the aliasing tag presented above.

3.3 Function and calls

I added functions to the formalisation. The main differences with procedures are that a function has a return value, then a call to a function is an expression, so it cannot have side effects in SPARK. Therefore, all parameters must be in mode `in` and variables declared outside the function cannot be assigned in the function body.

To do so, I changed the declaration of procedures so it includes a field `return_type` of type `maybe typ`, so functions can be declared with the same structure (procedures will just let this field empty).

Evaluating a function call is quite similar to a procedure call, except simpler, because the state is not modified after the call:

1. Fetch the declaration with its name in the symboltable
2. Evaluate the parameter values
3. Modify the symboltable so that all variables are turned to `in` mode

4. Select the frames that can be accessed inside the function body
5. Check the clock and decrease it
6. Evaluate the declarative part
7. Evaluate the statement part, it should return a `Ret_s v` value (that means that it reached a `return` statement)
8. Check the type of `v`, and return it

3.4 Type checking

To have the evaluating function behaving like the existing compiler GNAT, I added to the formalisation some dynamic type checking.

In SPARK, giving the type is required when declaring a variable: there is no type inference and the values given must strictly follow the expected type.

For example, in the following example, there would be a type error:

```
type small_int is new Integer range (0 .. 5);
X : Integer := 2;
procedure foo(z : in small_int) is ... end foo;
...
foo(X);
```

There the procedure wants a parameter of type `small_int`, which is defined to be a derived type of `Integer` between 0 and 5 (bounders included). Even if `X` has a value that fits in the `small_int` type, it is an `Integer`, so it would raise an error because the parameter given in `foo(X)` is not of the correct type.

But if the type declared is a `subtype` of another type, both types are compatible and it just has to follow the constraint of value to be correct. Example:

```
subtype small_int is Integer range 0 .. 5
```

I did a less powerful checking and in every cases, I just check that the value given is in the correct range to be the expected type. This check occurs when assigning a variable or when calling a procedure or a variable for the parameters.

For arrays and records, I also check that the value given is completely initialised with correct values and correct field names and indexes.

3.5 Loops

In the first formalisation, the only kind of loop that was available was `while`. But other loops exist in SPARK. There are `for` loops and also unlimited loops. There is also a statement `exit when x` that allows to exit any kind of loops when the condition `x` is fulfilled.

I added these different loops and this `exit` statement. The unlimited loop is evaluated as a `while true` statement.
The `for` loop, nevertheless could not be written as a `while` because the criteria of the loop (let's call it `i`) needs to be updated at every step of the loop, but cannot be assigned in the body loop, and such restrictions cannot be written in the `while` syntax. Then I did a specific semantic function to evaluate the for loop statement.

4 Flattening

I started to implement a compiler for SPARK. In this section and the next, I will summarise my progress on that.

4.1 Presentation

The first step for compiling was to flatten the SPARK AST, that is to say move all functions and procedures declaration to the top level and get rid of all nested declarations. Type declarations also need to be moved out, but not variables.

Example:

```
procedure f() is
  procedure g(x: in out Integer) is
    n: Integer := 0;
  begin
    n := n + 1;
    if x < Y then
      x := x + n;
      g(x);
      x := x + n;
    else x := 10;
  end g;
  X: Integer := 0;
  Y: Integer := 10;
begin
  g(X);
end f;
```

In this example, if the variable `n` was declared outside the procedure `g`, then after a recursive call, then there would be only one variable that has its value updated instead of having one variable for every call, the value of `n` before and after the call to `g` would be different, whereas it needs to stay constant.

But then, when moving the declaration of procedure `g`, the variable `Y` will not be declared anymore in its scope, so it could not be read. To prevent this kind of issue, one must add variables declared below the top level as parameters in the functions and procedures.

There is also a need to rename everything, because moving procedure and functions could cause names clashes. For example in the following example:

```
procedure f is
  procedure alias is begin ... end alias;
begin ... end f;
procedure g is
  procedure alias is begin ... end alias;
begin ... end g;
```

The both procedures `alias` are defined in different scopes so there is no confusion possible as it is written, but moving them will create an error so one can uniquely rename every objects in the program to prevent any error of this form.

4.2 Global description of the algorithm

First I declared a counter `c`, that will help enforce unique names: when something is declared, its name `n` is changed to `n_c`, then I keep a list of all combinations of old names to new names in order to modify every occurence of this name by the new unique one when reading the code.
The declarations are divided in two parts: the one that stays where they were (variable declarations only) and the one that are moved at the top level, above the procedure or function where they were (so it can still access it).
There is a need to keep all formal parameters and variables declared in the scope to know what to add

as parameters to the nested functions and procedures. When one is declared, it is added to a list of variables, with their types to change procedures signatures.
Example:

```
procedure f(x: out Boolean; y: in Integer) is
  a : Integer := 1;
  b : Integer;
  c : Boolean := false;
  procedure g(k: in out Integer) ... ;
  d : Integer := 2;
begin ...
end f;
```

In this example, in the procedure `g`, the variables `x, y, a, b, c` should be added as parameters. So `g` will look like the following (here the variables are not renamed, contrary to the real algorithm):

```
procedure g(k: in out Integer; x: in out Boolean; y: in Integer;
            a: in out Integer; b: in out Integer; c: in out Boolean) ...
```

4.3 Issue

After testing a bit, it appeared that this quite simple algorithm was not complete enough and that there were issues to tackle. I will present them now and how they were solved.

- The first issue is linked to the property of SPARK to forbid all possible aliasing. Then adding all variables as parameters could cause aliasing if these variables are also passed as regular parameters in the procedure or function. This is the second case of aliasing presented above, see 3.2.3.

  Example:

  Original program:

  ```
  procedure top is
    a : Integer := 2;
    b : Integer;
    procedure nested(y : in Integer;
            z : out Integer) is begin
      z := y + 1;
    end nested;
  begin
    nested(a, b);
  end top;
  ```

  Program after flattening:

  ```
  procedure nested_3(y_4 : in Integer;
  z_5 : out Integer; a_1 : in out Integer;
  b_2 : in out Integer) is begin
    z_5 := y_4 + 1;
  end nested_3;

  procedure top_0 is
    a_1 : Integer := 2;
    b_2 : Integer;
    begin
      nested_3(a_1, b_2, a_1, b_2);
    end top_0;
  ```

  There, the variables `a` and `b` are added as parameters to the `nested` procedures so it can still access to it even if the declaration is not in its scope anymore. But as these variables were passed as parameters, it causes aliasing.

  If `a` or `b` appeared in the first place in the procedure, there would have been aliasing (by the first way of created aliasing, between a global variable and a parameter, see 3.2.2), so this issue can be tackled by tracking which variable is really used in the call, and only add the ones that are useful. If there is aliasing, then there were already aliasing in the source code, so no error is created.

  In pratice, what I did is explore all potential branches and store for every procedure and function the variables they could use (by reading or writing).

Example:

```
procedure first is begin
  b : Boolean := false;
  x : Integer := 0;
  procedure second is begin
    x := x + 1;
  end second;
  procedure third(b : in Boolean) is begin
    if b then second;
    else null;
    end if;
  end third;
begin ... end first;
```

In this example, when flattening this program, it is not useful to add the variable `b` to the parameters of procedures `second` and `third`, to the contrary, `b` could be passed as a parameter for `third` but the variable `x` needs to be added in both, indeed this variable is needed in the `second` procedure, but as it may be called in `third` procedure, the variable `x` may be useful too.

- Then an other problem appeared concerning the mode of the parameters. In the first idea of the flattening algorithm, all variables added as parameters would have mode `in out` and mode `in` for functions, but this simple approach is not precise enough. Let's consider the following example:

```
procedure one is
  a: Integer := 0;
  b: Boolean := false;
  procedure plus1_or_minus1(x: in out Integer) is begin
    if b then x := x + 1;
    else x := x - 1;
    end if;
  end plus1_or_minus1;
  function example return Integer is
    n : Integer := 2;
  begin
    plus1_or_minus1(n);
    return n;
  end example;
begin
  b := true;
  plus1_or_minus1(a);
end one;
```

After the flattening pass (without renaming for clarity), it would look like:

```
procedure plus1_or_minus1(x: in out Integer; b: in out Boolean) is begin
  if b then x := x +1;
  else x := x - 1;
  end if;
end plus1_or_minus1;

function example(b : in Boolean) return Integer is
  n: Integer := 2;
begin
  plus1_or_minus1(n, b);
```

```
  return n;
end example;

procedure one is
  a: Integer := 0;
  b: Boolean := false;
begin
  b := true
  plus1_or_minus1(a, b);
end one;
```

The issue here is caused by the fact that when calling `plus1_or_minus1` in `example` the variable b is in mode `in` but the parameter needs a `in out` variable. So when copying back the value, it would raise an error because it tries to modify a variable whereas it is forbidden.

One cannot change the mode of the variable in the function call, because side effects in expressions are forbidden. However the parameter `b` could be in mode `in` in the procedure `plus1_or_minus1`, as it is not modified. If `b` was written in the procedure, the error would exist also in the source code, because the function `example` would have some side effects. Thus, for each procedure, each variable needs to be passed as a parameter with the right mode: a variable can either be read (`in`), written to (`out`), both (`in out`) or neither (the variable is removed from the parameters).

- Finally, passing parameters to `in` mode causes another error: the type checking pass added to the formalisation requires the values passed in parameters to be fully initialised (which is not the case for parameter in mode `out` or `in out` because they might be written before being read). For example, the following program would be accepted in SPARK but would be rejected after the flattening process, as the variable `A` is not completely initialised when used in another function:

```
type simple_array is array 0 .. 5 of Boolean;
procedure foo is
  A : simple_array;
  function read_array return Boolean is begin
    return A(0);
  end read_array;
begin
  A(0) := false;
  A(1) := read_array;
end foo;
```

This issue is a bit more complicated, as it cannot be solved only by modifying the flattening algorithm. It has not be solved yet but a solution could be to modify the type checking function to accept partially initialised or uninitialised variable if the expected type is correct.

5 Lexer and parser

5.1 Lexer

For the lexer (and the parser as well) I took the ones from the CakeML compiler as example for several reasons: it has been done in the same language (HOL4) and thus can be adapted quite easily and second, I could ask questions about the code to Michael Norrish that had been doing the CakeML lexer and parser.

The lexer is quite straightforward to do: it is mostly a function that takes the input as a string and separates all the "words" of the string. A "word" is composed of either a list of character, or a list of digits or a list of symbols (like `()[],;:` etc.).

Then there is just to identify each entity that has been found into the appropriate token:

- Digits are converted into base 10 numbers
- symbols are converted to their meaning (for example, the string `":="` is converted to the `AssignT` token)
- list of characters are either a keyword if it matches one (like `type`, `procedure` and so on), and is considered as an identifier elsewise.

5.2 Parser

5.2.1 Parsing expression grammar

To parse the language, I used a parsing expression grammar (PEG), that both allows to express simply the parsing rules of the language and that has a proved implementation in HOL4.

A PEG is quite similar to context-free grammar excepts that it is unambiguous. It includes the following constructions:

- Sequence between two parsing expressions
- Ordered choice between two parsing expressions
- Zero or more of one expression
- One or more of an expression
- Optional expression
- And-Predicate, it accepts only if the expression is accepted but it does not consume any input
- Not-Predicate, same as the And-Predicate but accepts only if the expression is rejected

This grammar is also used to parse the CakeML language and I took example of this parser to write the SPARK parser.

5.2.2 Application to SPARK

The HOL4 implementation works as follow: A peg is formed by a start rule and then a set of rules. A rule is written like `name_of_rule, rule` with the rule of type `pegsym`

Signature of pegsym type:

```
'a, 'b, 'c pegsym = empty 'c
| any ('a # locs) -> 'c
| tok ('a -> bool) (('a # locs) -> 'c)
| nt ('b inf) ('c -> 'c)
| seq pegsym pegsym ('c -> 'c -> 'c)
| choice pegsym pegsym ('c + 'c -> 'c)
| rpt pegsym ('c list -> 'c) | not pegsym 'c
```

Where `'a` is the type of input, here it is a list of tokens, `'b inf` designates a rule name and `'c` is the return type of the parser, that is fixed and constant for all rules.

The return type is then a list of parsed tree, that is defined as follow (already built in HOL4)

```
leaf = TOK ('a # locs) | INL ('b inf # locs)
parse_tree = Nd ('b inf # locs) (list parse_tree) | Lf leaf
```

So in the leaf, a token (or a rule name but in my parser it is always the token case that is used) is stored and in the node a rule name is stored. Then an other step needs to be done to convert the parsed tree obtained in the AST of the language. I cannot directly form the AST because of the requirement to have only one return type for all rules, whereas the AST is formed by several types.

In the implementation, there is less constructions than in the definition but it allows to express the same set of rules, for example the optional construction can be remade using the choice one. I then began to build a few intermediate function to have a simpler way to express the rules, for example the possibility to have choices or sequences of more than two elements, or the optional rule just mentioned.

I then wrote all the parsing rules that are useful in the subset of SPARK that I considered, these rules can be found in appendix B of this document.

5.2.3 Conversion to the AST

Once the parsing in itself is done, I have a parsing tree that I need to convert into the AST. The structure of the parse tree and the AST being quite similar, the conversion consists mostly in renaming the nodes to their equivalent in the AST, so it is quite straightforward.

One difficulty stands that the all process is written in HOL4, and that it needs to be proven at some point. The HOL compiler requires to have for example a termination proof of a function. These proofs had to be done for the lexer as well and if I managed to do some, I had not enough time to finish all of these so most of termination proofs are `cheated`, that is to say the proof is not done but the HOL4 compiler accepts it anyway. The functions can then be tested even if the proofs had not been done yet.

6 Further work

As this project was just at this beginning, the complete compiler has not been finished during the 5 month internship, I will there just do a list of a few things that still need to be done to complete the initial goal.

So concerning the compiler itself, there are several steps that still needs to be done, like the type checking of the AST or the conversion to the Pancake language. With the parsing that has been done and the flattening step to simplify the AST, that would be enough to have the compiler.

Furthermore, one significant requirement of the project that has not been done for the moment is the verification step. Having the compiler verified is what makes this project innovating and interesting and this is a large amount of work that remains here. The formalisation of the subset has still be done to prepare this step.

And finally, there are also a few things that can be improved, the different tasks I worked on has not been all completely finished and needs to be fixed (like the modification of the type checking for the flattening process explained in subsection 4.3) or the parsing has not been completed (in particular the conversion from the parsed tree to the AST).

7 Conclusion

To recap what I did in this internship, I first get familiar with the language that I had to compile, SPARK, and with the tools that I have been using for that, mainly *lem* and HOL4. Then I wrote a formalisation of a large subset of SPARK, based on an existing one, with some features added like checking for aliasing. After that I began to work on the compiler in itself by doing a step of simplification of the AST, called in this report flattening and then writing a lexer and parser for the language, which will simplify a lot the tests of the previous tasks.

I had never been doing formal proofs before and even if I have not done much proofs during the internship, I spent quite some time using HOL and it really interested me. I learned a lot during this internship, like the use of HOL4 and *lem* or how to read and find information in the reference manuals of the languages.

The subject of this internship really interested me, and it was very satisfying to stand at the beginning of the project. I had quite freedom in what I was doing and how for the different subgoals that I worked on. I am planning in a close future to continue a bit working on this project in my spare time, at least to finish and correct a few things on what I did, so this project can be easily taken by someone else.

References

[1] Oskar Abrahamsson, Anthony Fox Johannes Åman Pohjola, Alejandro Gómez-Londoño, Hrutvik Kanabar, Ramana Kumar, Andreas Lööw, Magnus Myreen, Michael Norrish, Scott Owens andThomas Sewell, Hira Syeda, Yong Kiam Tan, and Timotej Tomandl. Cakeml. *GitHub repository*. https://github.com/CakeML/cakeml.

[2] AdaCore. Mda selects adacore's gnat pro assurance development platform for international space station software. *AdaCore Press Releases*, 2011. https://www.adacore.com/press/mda-gnatpro-space-station.

[3] AdaCore. Siemens switzerland selects adacore toolset for railway project siemens-railway. *AdaCore Press Releases*, 2011. https://www.adacore.com/press/siemens-railway.

[4] AdaCore. Smartward pty ltd selects adacore tools for hospital information system development. *AdaCore Press Releases*, 2014. https://www.adacore.com/press/smartward-hospital-information-system.

[5] AdaCore. *Ada reference Manual*, 2016. http://docs.adacore.com/live/wave/arm12/html/arm12/arm12-TOC.html.

[6] AdaCore and Altran UK Ltd. *SPARK 2014 Reference Manual*. http://docs.adacore.com/spark2014-docs/html/lrm/index.html.

[7] HOL community. Hol. https://github.com/HOL-Theorem-Prover/HOL.

[8] Yannick Moy, Pierre Courtieu, and Rob Blanco. sparkformal. https://github.com/AdaCore/sparkformal.

[9] Dominic Mulligan, Kathryn E. Gray, Scott Owens, Peter Sewell, and Thomas Tuerk. lem. https://github.com/rems-project/lem.

[10] Dominic Mulligan, Thomas Tuerk, Scott Owens, Kathryn E. Gray, and Peter Sewell. *Lem Manual*. https://www.cl.cam.ac.uk/~pes20/lem/built-doc/lem-manual.html.

[11] Scott Owens, Magnus O. Myreen, Ramana Kumar, and Yong Kiam Tan. Functional big-step semantics. *esop*, 2016. https://cakeml.org/esop16.pdf.

[12] Zhi Zang, John Hatcliff, Yannick Moy, and Pierre Courtieu. Focused certification of an industrial compilation and static verification toolchain. *SEFM (Software Engineering and Formal Methods)*, 2017. https://blog.adacore.com/uploads/Focused-Certification-of-an-Industrial-Compilation-and-Static-Verification-Toolchain.pdf.

A Abstract Syntax Tree

```
type typ =
    | Boolean
    | Integer
    | Subtype string
    | DerivedType string
    | IntegerType string
    | ArrayType string
    | RecordType string
    | EnumType string

type unary_operator =
    | Unary_Minus
    | Not

type binary_operator =
    | And
    | Or
    | Equal
    | Not_Equal
    | Less_Than
    | Less_Than_Or_Equal
    | Greater_Than
    | Greater_Than_Or_Equal
    | Plus
    | Minus
    | Multiply
    | Divide
    | Modulus

type literal =
    | Integer_Literal integer
    | Boolean_Literal bool
    | Enum_Literal string

type mode =
    | In
    | Out
    | In_Out
```

```
type expression =
    | Literal literal
    | Name name
    | Binop binary_operator expression expression
    | Unop unary_operaor expression
    | CallFun string (list expression)

type name =
    | Identifier string
    | IndexedComponent name expression
    | SelectedComponent name string

type statement =
    | Null
    | Assign name expression
    | If expression statement statement
    | While expression statement
    | For string typ statement
    | Loop statement
    | Exit expression
    | CallProc string (list expression)
    | Seq statement statement
    | Ret expression

type rng = Range integer integer

type type_decl =
    | SubtypeDecl string typ rng
    | DerivedTypeDec lstring typ rng
    | IntegerTypeDecl string rng
    | ArrayTypeDecl string typ typ
    | RecordTypeDecl string list (string typ)
    | EnumTypeDecl string (list string)

type obj_decl = < |
    oname : string;
    otyp : typ;
    initialisation : maybe expression | >
```

$$
\begin{aligned}
&type\ param\_spec = < | \\
&\quad parameter\_name : string; \\
&\quad parameter\_type : typ; \\
&\quad parameter\_mode : mode\ | > \\
\\
&type\ decl = \\
&\quad | \ NullDecl \\
&\quad | \ TypeDecl\ type\_decl \\
&\quad | \ ObjDecl\ obj\_decl \\
&\quad | \ ProcBodyDecl\ proc\_body\_decl \\
&\quad | \ SeqDecl\ decl\ decl \\
\\
&type\ proc\_body\_decl = < | \\
&\quad p\_name : string; \\
&\quad p\_parameter\_profile : list\ param\_spec; \\
&\quad p\_return\_type : maybe\ typ; \\
&\quad p\_declarative\_part : decl; \\
&\quad p\_statements : statement\ | >
\end{aligned}
$$

B Parsing rules

$$
\begin{aligned}
\langle declaration \rangle &= (\langle type\_declaration \rangle \\
&\quad \mid \langle procedure\_declaration \rangle \\
&\quad \mid \langle function\_declaration \rangle \\
&\quad \mid \langle variable\_declaration \rangle) \ \texttt{;} \ \langle declaration \rangle ?
\end{aligned}
$$

$$
\begin{aligned}
\langle type\_declaration \rangle &= \langle subtype \rangle \mid \langle derivedtype \rangle \mid \langle integertype \rangle \mid \langle recordtype \rangle \mid \langle arraytype \rangle \\
\langle subtype \rangle &= \texttt{subtype}\ ident\ \texttt{is}\ \langle type \rangle \\
\langle derivedtype \rangle &= \texttt{type}\ ident\ \texttt{is new}\ \langle type \rangle \\
\langle integertype \rangle &= \texttt{type}\ ident\ \texttt{is range}\ \langle range \rangle \\
\langle recordtype \rangle &= \texttt{type}\ ident\ \texttt{is record}\ (\langle variable\_declaration \rangle\ \texttt{;})* \\
\langle arraytype \rangle &= \texttt{type}\ ident\ \texttt{is array}\ (\langle range \rangle \mid \langle type \rangle)\ \texttt{of}\ \langle type \rangle \\
\langle range \rangle &= \texttt{(}\ integer\ \texttt{..}\ integer\ \texttt{)} \mid integer\ \texttt{..}\ integer \\
\langle type \rangle &= ident\ (\texttt{range}\ \langle range \rangle)?
\end{aligned}
$$

$$
\begin{aligned}
\langle procedure\_declaration \rangle &= \texttt{procedure}\ ident\ \langle parameter\_list \rangle ?\ \langle procedure\_body \rangle \\
\langle function\_declaration \rangle &= \texttt{function}\ ident\ \langle parameter\_list \rangle ?\ \texttt{return}\ \langle type \rangle\ \langle procedure\_body \rangle \\
\langle procedure\_body \rangle &= \texttt{is}\ \langle declaration \rangle ?\ \texttt{begin}\ \langle statement \rangle\ \texttt{end}\ ident \\
\langle parameter\_list \rangle &= \texttt{(}\ ((\langle parameter \rangle\ \texttt{;})*\ \langle parameter \rangle)?\ \texttt{)} \\
\langle parameter \rangle &= ident\ \texttt{:}\ \langle mode \rangle ?\ \langle type \rangle \\
\langle mode \rangle &= \texttt{in out} \mid \texttt{in} \mid \texttt{out}
\end{aligned}
$$

$$
\langle variable\_declaration \rangle = ident\ \texttt{:}\ \langle type \rangle\ (\texttt{:=}\ \langle expression \rangle)?
$$

$$
\begin{aligned}
\langle statement \rangle &= (\langle assign \rangle \mid \langle if \rangle \mid \langle while \rangle \mid \langle for \rangle \mid \langle loop \rangle \mid \langle exit \rangle \\
&\quad \langle call \rangle \mid \langle ret \rangle \mid \langle null \rangle) \ \texttt{;} \ \langle statement \rangle ?
\end{aligned}
$$

$$
\begin{aligned}
\langle assign \rangle &= \langle variable\_name \rangle\ \texttt{:=}\ \langle expression \rangle \\
\langle if \rangle &= \texttt{if}\ \langle expression \rangle\ \texttt{then}\ \langle statement \rangle\ (\texttt{else}\ \langle statement \rangle)?\ \texttt{end if}
\end{aligned}
$$

$$
\begin{aligned}
\langle while \rangle &= \texttt{while}\ \langle expression \rangle\ \langle loop \rangle \\
\langle for \rangle &= \texttt{for}\ ident\ \texttt{in reverse}?\ \langle type \rangle\ \langle loop \rangle \\
\langle loop \rangle &= \texttt{loop}\ \langle statement \rangle\ \texttt{end loop}
\end{aligned}
$$

$$
\begin{aligned}
\langle exit \rangle &= \texttt{exit when}\ \langle expression \rangle \\
\langle ret \rangle &= \texttt{return}\ \langle expression \rangle \\
\langle null \rangle &= \texttt{null}
\end{aligned}
$$

$$
\begin{aligned}
\langle expression \rangle &= \langle and\_or\_op \rangle \\
\langle and\_or\_op \rangle &= \langle equal \rangle \ ((\texttt{||} \mid \texttt{\&\&}) \ \langle and\_or\_op \rangle)? \\
\langle equal \rangle &= \langle inequality \rangle \ ((\texttt{=} \mid \texttt{/=}) \ \langle equal \rangle)? \\
\langle inequality \rangle &= \langle add\_sub \rangle \ ((\texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=}) \ \langle inequality \rangle)? \\
\langle add\_sub \rangle &= \langle modulus \rangle \ ((\texttt{+} \mid \texttt{-}) \ \langle add\_sub \rangle)? \\
\langle modulus \rangle &= \langle product \rangle \ (\texttt{\%} \ \langle modulus \rangle)? \\
\langle product \rangle &= \langle unary\_operation \rangle \ ((\texttt{*} \mid \texttt{/}) \ \langle product \rangle)? \\
\\
\langle unary\_operation \rangle &= (\texttt{-} \mid \texttt{!})? \ \langle base \rangle \\
\\
\langle variable\_name \rangle &= ident \ (\langle record \rangle \mid \langle array \rangle)* \\
\langle record \rangle &= \texttt{.} \ ident \\
\langle array \rangle &= \texttt{[} \ \langle expression \rangle \ \texttt{]} \\
\\
\langle literal \rangle &= integer \mid \texttt{false} \mid \texttt{true} \\
\\
\langle call \rangle &= ident \ \texttt{(} \ ((\langle expression \rangle \ \texttt{,}) * \ \langle expression \rangle)? \ \texttt{)} \\
\\
\langle base \rangle &= (\texttt{(} \ \langle expression \rangle \ \texttt{)}) \mid \langle call \rangle \mid \langle variable\_name \rangle \mid \langle literal \rangle
\end{aligned}
$$