8/26/2022

Ada analysis in Rascal

Damien De Campos (ESI TNO)
Supervisors:
Pierre van de Laar (ESI TNO)
Jurgen Vinju (CWI SWAT TU/e)

Contents

1. Abstract

Nowadays software maintenance is an import part of software development. Technologies are fast moving and you need to upgrade your software if you want to stay competitive. Consumers always want better and faster software. Some part of software maintenance can be automated or semi-automated using analysis tools. The issue is that each languages has its own analysis tools and you have to learn different tools for each language that you want to analyze. That's the issue that Rascal tries to solved.

Rascal is a domain specific language for metaprogramming, such as static code analysis, program transformation, program generation and implementation of domain specific languages. It is a general meta language in the sense that it does not have a bias for any particular software language. Its syntax and semantics are based on procedural (imperative) and functional programming. It relies on front-ends to analyze software languages, one front-end is needed for each language. Some of the front-ends are already quite popular like the Java, C++, and PHP ones. However a lot of programming languages don't have a Rascal front-end yet.

One of these languages is Ada, a language that has been designed to meet the requirement of the US department of defense in the 1980s. It come up with a lot of features such as strong typing, oriented object programming, generics or exception handling. Ada has a reputation of being safe, that's why it's mainly used in the domain of defense, aerospace, transport, and finance.

At the time of writing this report if you want to analyze an Ada software you have few alternatives, the main one is Libadalang, a library of parsing and semantic analysis of Ada source code. There is also a tool called ASIS that is based on the gnat compiler but it is no longer supported. The last option is Rascal but we are missing an Ada front-end.

The goal of my 3 months internship is to build a Rascal front-end to enable Ada analysis. Instead of building my own parser that is a very complex task, this front-end is built on top of Libadalang. An already existing library well tested and popular. Libadalang is already delivered with multiple programming languages to manipulate its analysis result such as Ada, Python, C, and OCaml API. This front-end will provide a new way to leverage the Libadalang analysis. Since Rascal was designed for code analysis it will be easier to write analysis.

2. Introduction

2.1. Rascal

Rascal is a domain specific language for metaprogramming. Rascal integrates source code analysis, transformation and generation on the language level. It relies on front-end to analyze programing language.

Most Rascal front end works the same way, they rely on an external library, they don't use their own parser because that's a lot of works to write a full parser.

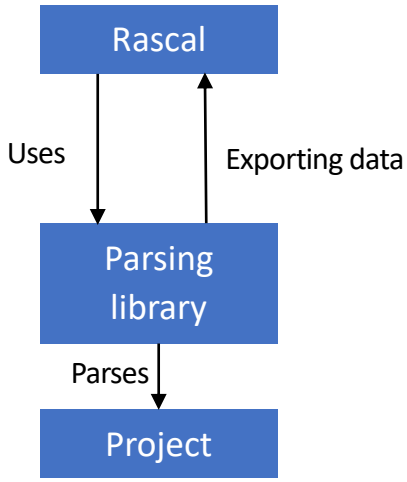


Figure 1: Generic front-end pipeline

Rascal provides a lot of types in its runtime like *int*, *real*, *bool*, *str* but also a lot of collection types like *list*, *set*, *map* or *relation*. There also more high level types like *location* (an URI), *dateTime* or *node*. The *node* type is used to build untyped tree structure. However Rascal allows you to declare your own algebraic data type to build typed tree structure. You can declare values of an algebraic data type by calling the constructor function. An algebraic data type can have multiple constructors.

The following data declaration defines the datatype *MyBool* that contains various constants (*tt* and *ff*) and constructors functions (*conj* and *disj*).

```
data MyBool = tt() | ff() | conj(MyBool L, MyBool R)  | disj(MyBool L, MyBool R);
```

Front-ends use their own data types to store the abstract syntax tree and one of the challenge of building a Rascal front-end is to convert the library AST representation into the Rascal representation. The Rascal module *ValueIO* is useful for exporting the data because it provides functions to deserialize Rascal data from string, binary stream or file.

Some algebraic data types generic to all programming language are already declared in the module *analysis::m3::AST* and they are named *Declaration*, *Expression*, *Statement* and *Type*. Front-ends that use these types have some analysis for free like the data flow analysis.

Once the data is imported in Rascal you can write your own analysis using the powerful features of Rascal. For example you can use the M3 model of Rascal. This model is a set of binary relations for programing

languages that are very useful for writing analysis because it contains semantic information. This model is very generic and can be used for all programming languages.

```
data M3(
    rel[loc name, loc src] declarations = {},
    rel[loc name, TypeSymbol typ] types = {},
    rel[loc src, loc name] uses = {},
    rel[loc from, loc to] containment = {},
    list[Message] messages = [],
    rel[str simpleName, loc qualifiedName] names = {},
    rel[loc definition, loc comments] documentation = {},
    rel[loc definition, Modifier modifier] modifiers = {}
) = m3(loc id);
```

You can see on the type above that it gathers data using locations to reference physical locations on files or logical locations like the name of declarations. Front-ends can also extend it to add new fields relative to the specificities of their language. Oriented object language might have the following fields in their m3 model: overrides, inherits, implements etc.

Rascal handles binary operation like addition, subtraction, multiplication or division in a simplistic way. Whereas in a lot of parsing tools they use some kind of abstraction of binary operators. An addition is often described in Rascal by a node *plus* with 2 fields: the 2 operand. That allows to easily identify all the additions of a program.

Rascal is an interpreted language, its interpreter is written in Java 11. There are different way to use Rascal:

- Eclipse plugin
- Vscode extension
- Command line (REPL) using a jar

If you choose the last option you need to use the following command:

```
$ java -Xmx1G -Xss32m -jar rascal-<version>.jar
```

2.2. Ada

Ada is a complex language with a lot of features. I will introduced you the Ada concepts relevant to this report.

An Ada project is made of a lot of packages, these packages are usually split into 2 files, the first one is the specification that described what is defined inside this package (like an header in C/C++), the second one is the implementation (also called the body) that contains the implementation of the package. In Ada there is a distinction between a function that returns a value and a procedure that doesn't. Subprogram is the word that is used to reference a function or a procedure.

Ada doesn't use bracket for scope like in a lot of programming language it uses *declare block* made of the keywords *declare* and *end*. These declare blocks separate declarations and statements, in other words you can't declare a variable everywhere. A declare block look like this:

```
declare
  X : Integer;
begin
  X := Integer'Value (Get_Line);
exception
  when Constraint_Error =>
    Put_Line ("Please enter a valid number");
end;
```

In the Ada code above we are declaring a variable, trying to convert the input string from the command line to an *Integer* to set the variable *X*. If the conversion fails it raises a *Constraint_Error* exception that is handled in the exception handler printing "*Please enter a valid number*".

In a declare block you can also defined a subprogram that will be visible only inside the declare block.

2.3. Libadalang

Libadalang is a library developed by AdaCore, the main company that maintained the Ada language. The Libadalang library allows users to analyze Ada software, it provides the abstract syntax tree (AST) and semantic queries on top of this tree. Users can write their own analysis using the programming language Ada, Python, OCaml, and C.

You can access each child (or field) of a node in the AST by using functions staring by *F_*. Sometime fields are optional because they are optional in the grammar. A simple example of that is when you declare an object the keyword *constant* is optional in the declaration. The function *F_Has_Constant* of object declaration in Libadalang can return *Constant_Present* or *Constant_Absent* depending if the keyword *constant* is present or not.

Libadalang used an oriented object style, *Ada_Node* is the root node, that means that all the other nodes used in the AST derived from it. Because of this structure, a lot of Libadalang nodes are abstract in the sense that they aren't not directly used in the AST, only the concrete nodes derived from abstract nodes are used. This implementation has a main drawback because some function can't specify with precision their returned nodes. Whenever a function can return 2 nodes that are drastically different, their common parent in the derived hierarchy might be *Ada_Node*, meaning that this function returns an *Ada_Node*. However, the node that is really returned by the function is one of the two nodes mentioned previously that derived from the root node.

Some nodes are a list of other nodes like: *Ada_Node_List*, *Basic_Decl_List*, *Compilation_Unit_List*…

Functions starting with *P_* are used to access properties, they often are semantic queries.

Libadalang provides an abstraction for binary operations, it uses a node named *Bin_Op* that has 3 fields: the 2 operands and the operation. Operation nodes are just empty nodes used to identify the operation.

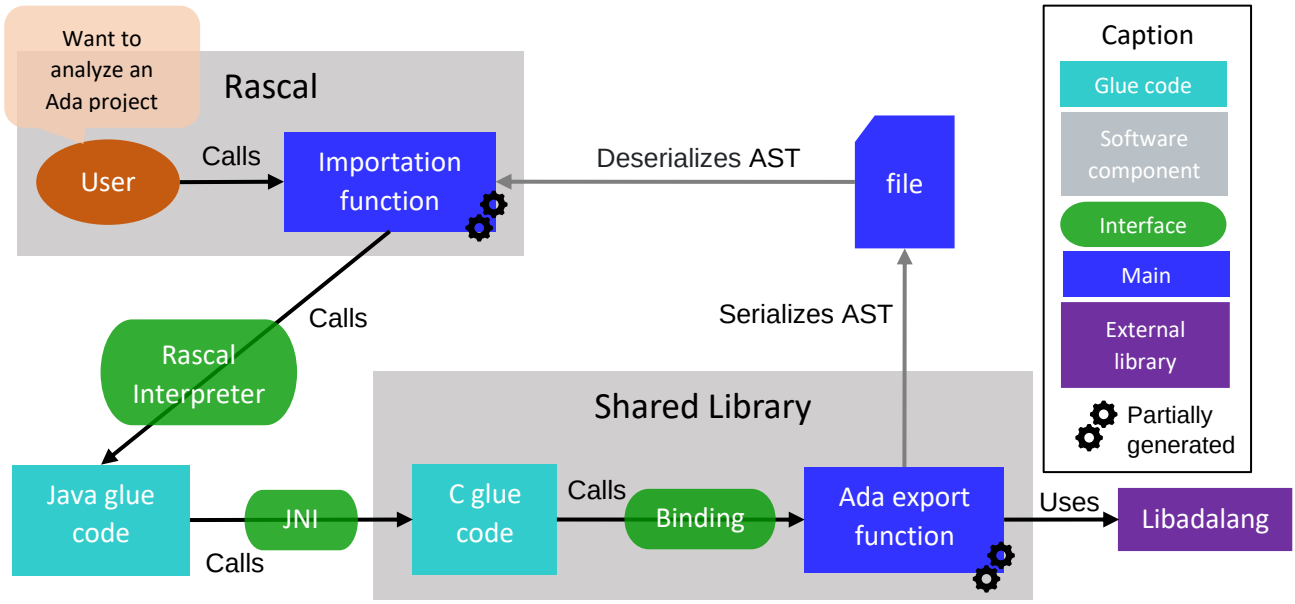
Libadalang is generated by a tools called Langkit. This tools, developed by AdaCore, takes in input an grammar and the specification of the nodes in the AST and generates a parser for the language.

3. Exporting data from Ada to Rascal

This project is named Ada-air for Ada analysis in Rascal. It's split into 2 main software components, the first one, written in Rascal, import an AST from a file, the second one, written in Ada, uses Libadalang to analyze a project and export the AST in a file.

These 2 components are partially generated by a Langkit plugin running during the generation of the Libadalang library.

The following picture describes how the front-end works. The next sections are explaining in details how the different part of this project works together.



3.1. Rascal data types

In order to export the AST produced by Libadalang in Rascal you need to defined some algebraic data types. These Rascal types are mapped on Libadalang nodes so that you don't need any post processing to export a Libadalang node into Rascal. Keep in mind that all these Rascal types are generated, more information in the section 4.1. For example the following Rascal type contains all the statements available in Ada (we have here only an extract of this type).

```
data Statement = call_stmt (Expression F_Call)
| assign_stmt (Expression F_Dest, Expression F_Expr)
| case_stmt (Expression F_Expr, list[Statement] F_Alternatives)
...
```

In order to ease the learning process of users that already know Libadalang, I used the same names. I expect that the future users of ada-air will be Libadalang users, that's why I kept the "*F_*". As you can see in the following figure, the *assign_stmt* constructor in Rascal has the same field name as the Libadalang node *Assign_Stmt*.

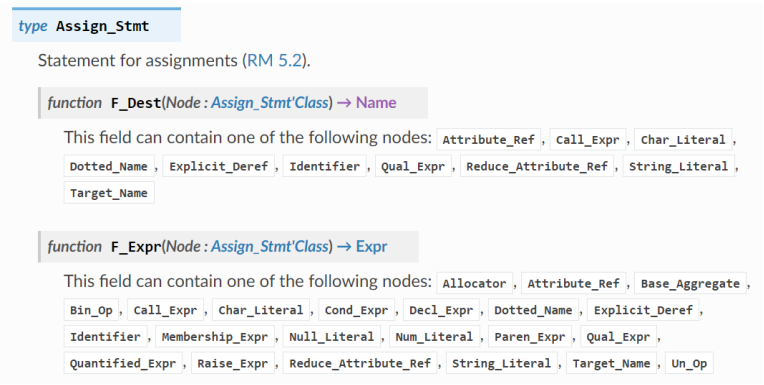


Figure 2 Libadalang documentation of Assign_Stmt

However, the Rascal algebraic data types aren't always a perfect mirror image of the Libadalang nodes. Some nodes aren't described in Rascal, other are describe in a different way.

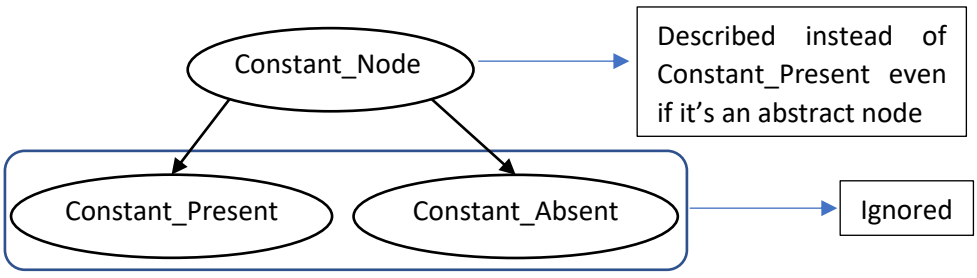
- Libadalang abstract nodes aren't describe because these types exist only for software architecture, they never appear in abstract syntax trees e.g., *Basic_Decl*.

- All the list nodes aren't describe too, instead we will simply use Rascal type *List*.

  Basic_Decl_List is described as *list[Declaration]*

- Libadalang nodes that end with *_Present* or *_Absent* (also named Boolean types in langkit) are described differently. Instead of using these types, we will use the Rascal list. An empty list for an absent node and a list containing one node for a present node. In that case we will use the parent type even if it's an abstract type because it makes more sense to named it without *_Present* if there is no *_Absent* node. To differentiate real list fields and boolean fields we are using a list alias with the name "*Maybe*".

  …_Absent are described as *[]*
  …_Present are described as *[Parent(…)]*



Four Libadalang nodes are inlined to simplify analysis in Rascal: *Bin_Op*, *Un_Op*, *Relation_Op*, and *Membership_Expr*. Libadalang provides an abstraction for operators but we don't want it in Rascal. We want to directly access the operation name.

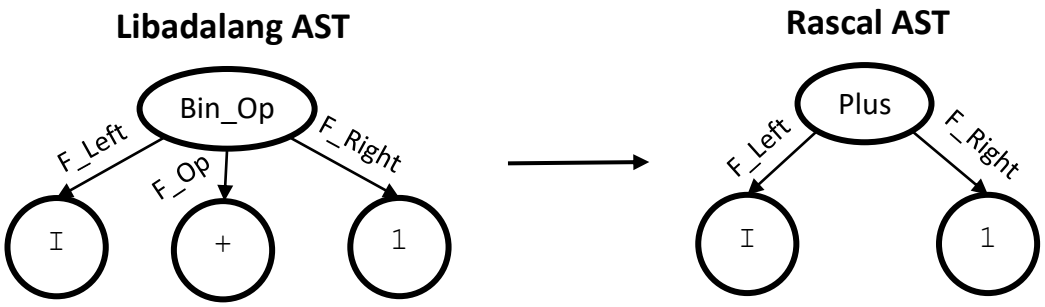


Figure 3 Difference between Libadalang and Rascal AST

However, we still need to be able to differentiate the 4 kind of operators, that's why the Rascal nodes that come from these Libadalang nodes have a prefix expect *Bin_Op* because it's the most used. Nodes that come from *Un_Op* are prefixed by *"Un_"*, the ones that come from *Relation_Op* are prefixed by *"Relation_"* and lastly the nodes that come from *Membership_Expr* are prefixed by *"Membership_"*. Only valid nodes are creating based on which operator these 4 nodes can contain.

Moreover these prefixes are also useful because multiple constructors with the same name aren't allowed in Rascal.

Some Libadalang nodes are merged together to reduce the number of generated Rascal types. For example, the Rascal type *Expression* contains the following Libadalang nodes : *Expr*, *Elsif_Expr_Type*, *Type_Expr*, *Others_Designator*.

Moreover, some Libadalang fields are *Ada_Node* because that's the common parent of all the kind of node that this field can contain. In Rascal *Ada_Node* is just a renaming of the untyped node which means that it can't be used otherwise this field will be untyped.

Example: the following Libadalang field is an *Ada_Node* because it can contain a *Qual_Expr* (derived from *Expr*) or a *Subtype_Indication* (derived from *Type_Expr*). Since these 2 nodes are merged into the Rascal type *Expression* it can be used in the Rascal constructor of Allocator instead of *Ada_Node*.

```
function F_Type_Or_Expr
     (Node : Allocator'Class) return Ada_Node;
   --  This field can contain one of the following nodes:
   --  * Qual_Expr
   --  * Subtype_Indication
```

Constructor of *Allocator* in Rascal:

```
allocator (Maybe[Expression] F_Subpool, Expression F_Type_Or_Expr)
```

However, some Libadalang *Ada_Node* fields can contain nodes that are too different to be merged into a single Rascal type. In that case we use a chain rule to avoid using *Ada_Node*.

For example, if an *Ada_Node* field can contain a *Statement* or a *Declaration* I need to use the following chain rule:

```
data Stmt_Or_Decl(loc src=|unknown:///|) = decl_kind(Declaration As_Decl)
| stmt_kind(Statement As_Stmt);
```

- If the field contains a *Type_Decl* I will use *decl_kind (type_decl (…))*.
- If the field contains an *Assign_Stmt* I will use *stmt_kind (assign_stmt (…))*.

Later, while manipulating the AST in Rascal, if you come across this chain rule, you can check whether it's a *Declaration* or a *Statement* and get the underlying node.

Each constructor of an algebraic data type has named fields, but the same field name may only be used in different constructors if it has the same type. Due to the merging of lots of different Libadalang nodes into the same Rascal type it occurs that the same field name is used in different constructors with different type.

In order to generate valid Rascal code, a python function of the Langkit plugin checks for name collision and renames the problematic fields to avoid this issue before writing down the types. There are different renaming rules that are not very robust if there are a lot of collisions but it works fine for now. That's maybe somethings to improve in the future.

The renaming rules are presented below, *F_X* is the original field name (always starting with "*F_*" like in Libadalang). They are ordered in priority, the first one is the most used and the last one should be avoided:

Original field	Renamed field
`list[A] F_X`	`list[A] F_X_List`
`maybe[A] F_X`	`maybe[A] F_X_Maybe`
`A F_X`	`A F_X_A`
`A_B F_X`	`A_B F_X_B`
`Ada_Node F_X`	`Ada_Node F_X_Node`
`X F_X`	`X F_X_X`

3.2. Deserializing a Rascal file

The Rascal types described previously are used to parse a Rascal file. This text file contains a representation of the Ada AST using constructors of the algebraic data types. All these constructors nested together are used to build the tree in Rascal.

The function used to read this file is declared in the Rascal runtime, more precisely in the module *ValueIO* that provides functions for reading and writing values in textual and binary format.

Each language handled by Rascal has all its modules starting with *lang::* followed by the name of the language and finally with the name of the module.

The Rascal front-end has 2 modules, the first one, *lang::ada::AST*, declares all the algebraic data types, the second one, *lang::ada::ImportAST,* declares functions to import the AST of an Ada file or all the AST of an Ada project.

3.3. Ada library

The challenge of this project is exporting the data gathered using Libadalang into a Rascal file. Some analysis rely on Libadalang built in properties but some analysis are ad-hoc. The Ada library in the folder *src/main/ada/* takes care of gathering the data and printing it into a Rascal file. Functions defined in *lang::ada::ImportAST* call this Ada library (more information in section 3.5) and parse the generated Rascal file.

The Ada library is split in 4 folders:

- *Glue-code*: holding the C functions and some Ada wrapper functions that translate C data types to Ada data types (see section 3.5 for details).
- *Export*: holding the code that export the AST into a file.
- *M3*: holding functions that returns m3 extra data used during the AST export.
- *Utils*: holding a package for wide wide string manipulation

The generated function of the Ada library is the one that exports the AST in Rascal by writing the constructors that we have seen before into a file. It's a recursive function that visits all the AST nodes and for each one writes the associated Rascal constructor. This function was causing stack overflow on huge file. The option *-Xss32m* increases the stack size, it's mentioned in the Rascal main web site, you must use it, otherwise you will get stack overflow. You should increase the stack size with this option if you are facing stack overflow issues.

In the Ada library W*ide_Wide_String* are used everywhere because I was afraid to not handle utf-16 and other characters encoding but I think that was a bad idea and I should use classic string anyway because Libadalang seems to already handle characters encoding properly.

3.4. Export function

The export function is a recursive function with a large case statement that handles each node independently. For each node we know what its fields are, whether they are optional or whether they need a chain constructor like `Stmt_Or_Decl`. It's important to understand that it's the parent node that determines if a node is optional are if it need a chain constructor. The same node can be used in different fields in different parents and sometimes being optional and sometimes not.

We will see how a node is serialized into Rascal. I took the example of a record component list.

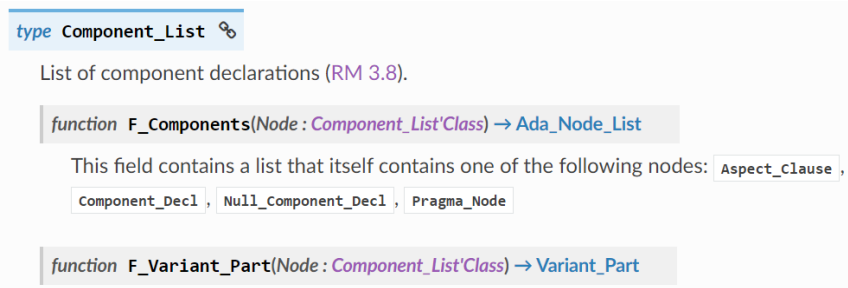


Figure 4 Libadalang documentation of Component_List

We can see in the documentation that a *Component_List* has 2 fields: *F_Components* and *F_Variant_Part*. The first one is an *Ada_Node_List* that can contains statements (*Aspect_Clause* and *Pragma_Node*) and declarations (*Component_Decl* and *Null_Conponent_Decl*). For this field we will use the chain constructor *Stmt_Or_Decl*. What the documentation doesn't tell us but that we know thanks to Langkit is that these 2 fields are optional.

How the export function handle *Component_List*:

```
when LALCO.Ada_Component_List =>

  Put (F, Opt);

  Put (F, "component_list(");

  Export_Ast (Print_Context => Context.Add_Indent_Level (Print_Context),
              Type_Context => (N                        => N.F_Components,
                               Is_Optional              => True,
                               Need_Chained_Constructor => True));

  Put (F, ",");
  Export_Ast (Print_Context => Context.Add_Indent_Level (Print_Context),
              Type_Context => (N                        => N.F_Variant_Part,
                               Is_Optional              => True,
                               Need_Chained_Constructor => False));


  Put (F, ",");
  Put (F, M3.Analysis.Get_Src_Annotation (N));
  Put (F, ")");
  Put (F, End_Opt);
```

Printing "[" if this node is optional

Constructor name of this node

Recursive calls for each fields

Optional field that requires chain constructor

Optional field that doesn't require chain constructor

Printing m3 fields

Closing all opened parenthesis, bracket…

The *Opt* and *End_Opt* variables are matching brackets, respectively *"["* and *"]"* if the current node is optional otherwise they are empty string.

The *Component_List* node doesn't need a chain constructor but we can take a look at the node *Null_Component_Decl* that can require a chain constructor. For these nodes, there is one if statement before and after the serialization of the node to add the additional constructor.

```
when LALCO.Ada_Null_Component_Decl =>
 Put (F, Opt);
 if Type_Context.Need_Chained_Constructor then
   Put (F, "decl_kind(");
 end if;
 Put (F, "null_component_decl(");
 Put (F, M3.Analysis.Get_Src_Annotation (N));
 if Type_Context.Need_Chained_Constructor then
   Put (F, "),");
   Put (F, M3.Analysis.Get_Src_Annotation (N));
 end if;
 Put (F, ")");
 Put (F, End_Opt);
```

Additional constructor

All the other functions of this library aren't generated because they used stable part of Libadalang that might not change in the future. Moreover they have a very complex logic that's hard to generate. It also makes the plugin simpler by generating less code.

Rascal uses a lot of URI through the location type, those URI can be physical like a path to a file or logical like the name of a function. The library has a package that handles the creation of Rascal URI. It allows the creation of a physical location from an Ada_Node by using its *Source_Location_Range* or by creating a logical location from a pre-built *String*. All the characters aren't allowed in an URI like space, quotes, less than sign, more than sign etc. These characters are escaped using their hexadecimal value.

Exporting the AST in Rascal was "easy" because data already existed. One of the hardest part of this project is creating the M3 model. The Ada library add extra data into the AST, this data will be later collected in Rascal to build the M3 model. This data is the result of ad-hoc analysis.

Libadalang doesn't provides a property that returns an unique name for a declaration, therefore the Ada library has its own ad-hoc analysis. For example, you can declare multiple variables with the same name in different scopes. To assure unique names I took my inspiration from the Java analysis in Rascal that specify the scope of the declarations.

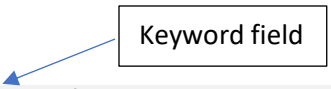
```
package body P is
  procedure Test is
  begin
    declare
      My_Var : Integer;      <-------  P/Test/scope(0)/My_Var
    begin
       ...
    end;
    declare
      My_Var : Integer;      <-------  P/Test/scope(1)/My_Var
    begin
       ...
    end;
  end Test;
end P;
```

Ada allows subprogram overloading, so I need also to differentiate 2 functions with the same name but different signatures:

```
package P is
  function Test (A : Integer)          <-------  P/Test(Integer):String
                return String;

  function Test (A : Float)            <-------  P/Test(Float):String
                return String;
end P;
```

To export these data in Rascal we are using the concept of keyword fields. These fields are not attached to a constructor but directly to the data type, so you can set these fields using any constructors.




```
data Assoc(loc src=|unknown:///|) = aspect_assoc (Expression F_Id, Maybe[Expression]
F_Expr_Maybe);
```

For now, I'm exporting only 3 fields of the M3 data type:

- declarations: maps declarations to where they are declared.
- containment: what is logically contained in what else.
- uses: maps source locations of usages to the respective declarations.

However, only these 3 fields are already very useful for writing analysis. For example using the declarations and uses fields you have cross-references and you can jump from a reference to its declaration. The C++ front-end doesn't fill all the M3 fields but it's already used a lot.

After exporting the AST in Rascal, a function visits all nodes and collects all these keyword fields to build the M3 model.

3.5. Calling Ada code from Rascal

To start your analysis of Ada source code you first need to import the AST in Rascal. To do so you have to use the module *lang::ada::ImportAST* that defines 2 functions, depending on whether you want to import a single file or a project:

```
Entry_Point importAdaAST(loc file)
map[loc file, Entry_Point Unit] importAdaProject(loc file)
```

In Rascal you can easily call Java methods using the following syntax to import a method:

```
public java ReturnType MethodName(ArgumentType ArgumentName, ...);
```

A Java class named *ImportAst* publicly exposes 2 methods that are called by the Rascal functions. I can now call native code from Rascal using the Java Native Interface. The class loads a shared library and calls 2 functions from it.

C function called from Java:

```
extern const char* Ada_Export_Project_Wrapper(const char* ada, const char* out);

JNIEXPORT void JNICALL Java_lang_ada_ImportAst__1importAdaProject
  (JNIEnv *env, jobject thisObj, jstring adaFileName, jstring outFileName) {
  const char *ada = (*env)->GetStringUTFChars (env, adaFileName, NULL);
  const char *out = (*env)->GetStringUTFChars (env, outFileName, NULL);
  const char* e = Ada_Export_Project_Wrapper (ada, out);
  if (e != NULL)
    {
      jclass Exception = (*env)->FindClass(env, "lang/ada/AdaException");
      (*env)->ThrowNew(env, Exception, e);
    }
}
```

The function *GetStringUTFChars* converts Java strings into C strings. The Ada wrapper function is called with the 2 strings decoded. The value returned by this function is a string containing the message of the exception if an exception occurred in Ada. If the string isn't empty I'm raising a Java exception called *AdaException*. That allow the Rascal user to catch it by doing:

```
try
  U = importAdaAST(f);
catch JavaException(class, msg): // if an exception occurred in Ada
  println (class + " " + msg);
catch IO(msg):  // if an exception occurred when opening the file in Rascal
  println ("IO: " + msg);
```

4. Building the project

4.1. Generation

Libadalang is generated by Langkit, a tools developed by AdaCore. This tools provides a way to a add plugins that will be run after the generation of Libadalang. That means that the plugin has access to the data used to generate Libadalang so you can easily generate other features.

In this internship a Langkit plugin that generates the Rascal algebraic data types and the Ada function that export the AST was developed.

Langkit and the plugin use a template library named mako. This library provided templates with control statements that facilitated the generation of source code.

The first advantage of generating these 2 software components is that it is easy to move to a new version of Libadalang, you just need to relaunch the plugin. The others piece of code that aren't generated are less affected by the changes of the Libadalang API because they used only stable part of the API. The second advantage is that the generated code is very repetitive so it will be error prone to manually write it.

4.2. How to setup

You need to first clone the project ada-air with submodules:

```
$ git clone --recurse-submodules https://github.com/cwi-swat/ada-air.git
```

In order to generate the missing files and build everything you need Python 3.9 or 3.10, pip, Java (JDK version 11), alire, and maven installed and the JAVA_HOME environment variable of set to the default location. You can then run the install script (bash or powershell) based on your OS.

The script first checks that the JAVA_HOME variable is set otherwise it will failed compiling the Ada library.

Then, the first command launched by the script is cloning the branch 22.0 of Langkit in a temporary directory and installing it with pip. The version of Langkit installed must be the same as the Libadalang version of the submodule and the one specify in the alire.toml file. At the time of writing this report the newest version available in alire is the 22.0.

```
$ pip install -U git+https://github.com/AdaCore/langkit.git@22.0
```

The next line launches the generation of the Libadalang library together with the langkit plugin.

```
$ PYTHONPATH=./src/langkit-plugin/ python ./Libadalang/manage.py generate --plugin-
pass=rascal_plugin.RascalPass
```

The last 2 commands are building the Ada shared library using Alire in the ada directory and compiling the Java classes using Maven in the root directory. These 2 package managers will download the missing dependencies.

```
$ cd src/main/ada/; alr build; cd -
$ mvn compile
```

5. Example

The following example have been tested on Zip-Ada, an open source library for dealing with zip compressed archive. This library has been chosen because of its small size, only 25 000 lines of code.

5.1. Cyclomatic Complexity

A simple example of a Rascal script computing the cyclomatic complexity of all subprograms (function and procedure). Cyclomatic complexity is a software metric used to indicate the complexity of a program by measuring the number of linearly independent paths through a program's source code.

Importing the project and looping over all subprograms:

```
void main(list[str] args) {
    loc project = |file:///| + args[0];
    map[loc,Entry_Point] Units = importAdaProject (project);
    for(file <- Units)
        for(/subp_body(_, spec, _, _, stmts, _) <- Units[file]) {
            int c = computeCC(stmts) + 1;
            str subp_name = Get_Name(head(spec.F_Subp_Name).F_Name);
            println("<subp_name> : <c>");
        }
}
```

This for loop will found all the subprogram implementation, *subp_body*, (we can also look for task body and expression function) that are in the AST. The following function visits all the subprogram statements and increases the complexity if there is a new path. The visit statement goes through all the nodes and tries to match every node with one case. The first 2 case statements subtract the complexity of their children so that it doesn't increase the complexity when the visit statement go through the children in order to not taking into account exception handler and nested subprogram. All the other case statements increase the cyclomatic complexity according to the number of branches that the matched node creates.

```
int computeCC(value N) {
    int c = 0;
    visit(N) {
        case s:exception_handler(_,_,_):  c -= computeCC(getChildren(s));
        case s:subp_body(_,_,_,_,_,_):   c -= computeCC(getChildren(s));
        case for_loop_stmt(_,_,_):   c += 1;
        case while_loop_stmt(_,_,_):  c += 1 ;
        case if_stmt(_,_,alternatives,_):  c += 1 + size(alternatives);
        case case_stmt(_,alternatives): c += size (alternatives) - 1 ;
        case if_expr(_,_,alternatives,_):  c += 1 + size (alternatives);
        case case_expr(_,alternatives):  c += size (alternatives) - 1;
        case exit_stmt(_,cond): c+= size (cond);
        case and_then(_,_):  c += 1;
        case or_else(_,_): c += 1;
        case quantified_expr(_,_,_): c += 2;
```

```
        case select_stmt(Alts,Elses,Abort): {
            c += size(Alts) - 1;
            c += if(isEpmty(Elses)) 0; else 1;
            c += if(isEpmty(Abort)) 0; else 1;
        }
    }
    return c;
}
```

We can compare the result with the tool gnat metric from AdaCore that also computes, among other things, the cyclomatic complexity.

Ada-air	gnat metric
HufT_build : 39 Repack_contents : 37 UnZipAda : 36 Guess_type_from_name : 35 Process_argument : 34	HufT_build (procedure body at lines 69: 360) === Complexity metrics === cyclomatic complexity : 39 Repack_contents (procedure body at lines 660: 1327) === Complexity metrics === cyclomatic complexity : 37 UnZipAda (procedure body - library item at lines 28: 348) === Complexity metrics === cyclomatic complexity : 36 Guess_type_from_name (function body at lines 307: 384) === Complexity metrics === cyclomatic complexity : 35 Process_argument (procedure body at lines 200: 302) === Complexity metrics === cyclomatic complexity : 34

Figure 5: The five most complex subprogram of Zip-Ada

We can see that the results are the same, however the main advantage of using rascal is that you can tailored the analysis to fit your needs. Moreover the result is easier to reuse in further analysis. Gnat metric generates a metric file for each Ada file, to reuse its analysis you need to manually post process all the generated files.

6. Tests

It's not an easy project to test because Ada is a very large programming language with a lot of features. Therefore, it's hard to think about all the combination possible in order to be sure to handle all the corner cases.

The first method of testing used is robustness tests, we tries to parse a lot of different style of Ada code: generated code, low-level code, parallelized code… With different library such as Libadalang, Langkit, Ada Web Server, Gnatcoll, xmlada, Ada Drivers Library… That represent nearly 1.5 million of lines of code which cover more that 90% of all the Rascal constructors. That makes sure the bridge is robust because it's able to parse all this code without crashing.

Some properties of the AST are checked in Rascal using the following functions:

- All the nodes need to have a *src* keyword field

```
bool allNodesHaveASource(node N) = (true | it && c.src? | /node c <- N);
```

- All the children of all the nodes must appear in the same order as they appear in the file

```
bool allNodesAreOrdered(node N) { ... }
```

These functions are new and they are going to be used to test other front-ends.

However, that's far away of being enough, we need to be sure that the analysis is correctly exported, especially the M3 model. That's the hardest part because it required to write analysis in Rascal and compare the result with another analysis software or with the same analysis written with another tools.

7. Ways of improvements

Improving tests: for now I mostly test the AST export and not the M3 model that is harder to test because you need to write complex analysis to be sure that the M3 model is robust and enough complete.

Another step will be to improve the m3 model that is exported. We can for example add the missing fields like messages, types, names, modifiers. We can also extend the M3 model to add new fields related to Ada specificities.

We can also use other types that are available in the m3 library like *TypeSymbol*, *Modifier* etc. This will ease the writing of analysis.

8. Future work

The biggest step will be to integrate the way back, in other words be able to rewrite the source code that you are analyzing. This will drastically improve the user experience and the possible uses of the front-end.

9. Contributions to the community

Issues and bugs that was found in Rascal during this project:

- No Java method for getTraversalContext · Issue #1620 · usethesource/rascal (github.com)
- Anchor links broken in docs.rascal-mpl.org · Issue #1619 · usethesource/rascal (github.com)
- ValueIO functions can't read Maybe field · Issue #1615 · usethesource/rascal (github.com)
- Rascal path function gives invalid windows path · Issue #1613 · usethesource/rascal (github.com)
- Documentation not up to date https://www.rascal-mpl.org/start/ · Issue #1606 · usethesource/rascal (github.com)

Bug in Libadalang that was found in this project:

- Missing documentation on some F_... functions · Issue #945 · AdaCore/libadalang (github.com)

10. Conclusion

This project is a success, we are able to analyze Ada software using Rascal in only 3 months.

Building a front-end for a complex language such as Ada is not trivial, especially if you don't know Rascal or the language that who want to build a front-end for. Already knowing them (Rascal and your language) helps a lot. Knowing the language that you want to write a front-end for is better than knowing Rascal because you need to handle every feature that is available in that language. However, you will not use all the features of the Rascal language.

If you are building your front-end using an already existing parser you need a good understanding of how it works. Exporting the AST is the "easiest" step but also the most important because this step is the heart of the front-end. It's the first building block of the rest of the front-end. To me the hardest part is to build the M3 model because you need to write complex and robust analysis in order to have correct information. You can write these analysis in Rascal using only the AST or using the library/parser that you are using if it offers these kind of features. The choice is up to you.

Libadalang is still the way to go for building complex and robust analysis because it's more stable than the Rascal front-end. Powerful queries are available in Libadalang that are not available yet in Rascal. Many examples exist of analysis written with Libadalang from which you can take your inspiration.

Rascal is the way to go if you want to write simple analysis where you can do some assumptions to simplify your script. Your script will maybe be less robust but it will be quicker to write. I think that can be a viable alternative of the Python binding provided by Libadalang.

I found Racal hard to learn because you can often write code that works but isn't written in a "Rascal way". In Rascal you can often write one line function and expression that does a lot of things by combining collection comprehensions and data queries.

I have deepened my knowledges about Libadalang, even if I had worked for 2 years with Libadalang before starting this internship I still learnt a lot. Building a front-end requires knowledge about all the features of the language however I worked mostly with the Ada type system during these 2 years.

I also discovered Rascal that is an amazing project that I will certainly use in the future.