Easy Ada tooling with Libadalang

Pierre-Marie de Rodat Raphaël Amiard

Software Engineers at AdaCore

The need

In three bullet points

- A library that allows users to query data about Ada sources
- Syntactic and semantic information
- Both low level and high level information
- Should be easy to integrate into tools/IDEs from different platforms/languages/etc.

The need - IDEs

```
174     end record;
175
176     type Cond_Branch_Context is limited record
177        Decision_Stack : Decision_Occurrence_Vectors.
178        --  The stack of open decision occurrences
179
180        Basic_Blocks   : Basic_Block_Sets.Set;
181        --  All basic blocks in the routine being ana
182
183        Stats          : Branch_Statistics;
184        --  Statistics on conditional branches in the
185
186        Subprg         : Address_Info_Acc;
187        --  Info of enclosing subprogram
188     end record;
189
190     procedure Analyze_Routine
191       (Name   : String_Access;
```

Figure 1: Syntax & block highlighting

The need - IDEs

Figure 2: Cross references

The need - IDEs

```
 8   function "+"
 9     (S1, S2 : Unbounded_String)
10      return Unbounded_String is
11     (S1 & "." & S2);
12
13   function Concat (Ns : String_Array) return String
14   is
15      R : Unbounded_String;
16   begin
17      for N of Ns loop
18         if Length (R) = 0 then
19            R := N;
20         else
21            R := R & N;
22         end if;
23      end loop;
24   end Concat;
```

```
 8   function Concat
 9     (S1, S2 : Unbounded_String)
10      return Unbounded_String is
11     (S1 & "." & S2);
12
13   function Concat (Ns : String_Array) return String
14   is
15      R : Unbounded_String;
16   begin
17      for N of Ns loop
18         if Length (R) = 0 then
19            R := N;
20         else
21            R := Concat (R, N);
22         end if;
23      end loop;
24   end Concat;
```

Figure 3: Refactoring

The need - command line tools

```
procedure Main is
   type my_int is new Integer range 1 .. 10;
   Var : my_int := 12;
begin
   null;
end Main;
```

```
$ ./my_custom_lal_checker main.adb
main.adb:2:9: Variable name should start with lowercase letter
main.adb:3:4: Type name should start with uppercase letter
```

Why not ASIS/GNAT?

Challenges

- Incremental: don't recompute everything when the code changes
- Error recovery: ability to compute partial results on incorrect code
- Long running: be able to run for 3 days without crashing your machine

GNAT and AdaCore's ASIS implementation are ill suited to those challenges.

Design goals

- Multi-language: Easy binding generation to other languages/ecosystems
  - Today: Python, Ada, C
- Easy scripting: Be able to create a prototype quickly & interactively
- Both low & high level APIS:
  - What is the type of this expression?
  - How many references to this variable?
  - Give me the source location of this token
  - Rename this entity
  - Etc.

API Part 1: Tokens

```
--  main.adb
procedure Main is null;
```

```
ctx = lal.AnalysisContext()
unit = ctx.get_from_file('main.adb')
for token in unit.root.tokens:
    print 'Token: {}'.format(token)
```

Outputs:

```
Token: <Token Procedure u'procedure' at 1:1-1:10>
Token: <Token Identifier u'Main' at 1:11-1:15>
Token: <Token Is u'is' at 1:16-1:18>
Token: <Token Null u'null' at 1:19-1:23>
Token: <Token Semicolon u';' at 1:23-1:24>
```

API Part 2: Syntax

```
procedure Main is
   A : Integer := 12;
   B, C : Integer := 15;
begin
   A := B + C;
end Main;
```

```
for object_decl in unit.root.findall(lal.ObjectDecl):
    print object_decl.sloc_range, object_decl.text
```

Outputs:

```
2:4-2:22 A : Integer := 12;
3:4-3:25 B, C : Integer := 15;
```

API Part 3: Semantic

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
    function Double (I : Integer) return Integer is (I * 2);
    function Double (I : Float) return Float is (I * 2.0);
begin
    Put_Line (Integer'Image (Double (12)));
end Main;
```

```
double_call = unit.root.find(
    lambda n: n.is_a(lal.CallExpr) and n.f_name.text == 'Double'
)

print double_call.f_name.p_referenced_decl.text
```

Outputs:

```
function Double (I : Integer) return Integer is (I * 2);
```

API Part 4: Tree rewriting (not finished yet!)

```
procedure Main is
begin
    Put_Line ("Hello world");
end Main;
```

Let's rewrite:

```
call = unit.root.findall(lal.CallExpr) # Find the call
diff = ctx.start_rewriting() # Start a rewriting
param_diff = diff.get_node(call.f_suffix[0]) # Get the param of the call
# Replace the expression of the parameter with a new node
param_diff.f_expr = lal.rewriting.StringLiteral('"Bye world"')
diff.apply()
```

Outputs:

```
procedure Main is
begin
    Put_Line ("Bye world");
end Main;
```

An example

```
import sys
import libadalang as lal

def check_ident(ident):
    if ident.text[0].isupper():
        print '{}:{}: variable name "{}" should be capitalized'.format(
            ident.unit.filename, ident.sloc_range.start, ident.text
        )

ctx = lal.AnalysisContext()
for filename in sys.argv[1:]:
    u = ctx.get_from_file(filename)
    for d in u.diagnostics:
        print '{}:{}'.format(filename, d)
    if u.root:
        for decl in u.root.findall(lal.ObjectDecl):
            for ident in decl.f_ids:
                check_ident(ident)
```

Technical prototypes/demos

Syntax highlighter/Xref explorer

```
with Ada.Text_IO; use Ada.Text_IO;
use all type Ada.Text_IO.File_Type;

procedure Example is

   subtype Nat is Integer range 0 .. Integer'Last;

   type Rec (N : Natural) is tagged record
      S : String (1 .. N);
   end record;

   type Money_Type is delta 0.01 digits 14;

   generic
      with procedure Put_Line (S : String);
   package Things is
      procedure Process (S : access Wide_String)
        with Pre => S /= null and then S'Length > 0
                    and then (for all I in S.all'Range =>
                                 S.all (I) / ASCII.NUL);
   end Things;
```

Figure 4: Libadalang based highlighter

Syntax based static analyzers

```
def has_same_operands(binop):
    def same_tokens(left, right):
        return len(left) == len(right) and all(
            le.is_equivalent(ri) for le, ri in zip(left, right)
        )
    return same_tokens(list(binop.f_left.tokens), list(binop.f_right.tokens))

def interesting_oper(op):
    return not op.is_a(lal.OpMult, lal.OpPlus, lal.OpDoubleDot,
                       lal.OpPow, lal.OpConcat))

for b in unit.root.findall(lal.BinOp):
    if interesting_oper(b.f_op) and has_same_operands(b):
        print 'Same operands for {} in {}'.format(b, source_file)
```

Those 20 lines of code found 1 bug in GNAT, 3 bugs in CodePeer, and 1 bug in GPS (despite extensive testing and static analysis).

More info on our blog

Semantic based static analyzers

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Main is
   Input : File_Type;
begin
   Open (File => Input, Mode => In_File, Name => "input.txt");

   while not End_Of_File (Input) loop
      declare
         Line : String := Get_Line (Input);  <--- WARNING: File might be closed
      begin
         Put_Line (Line);
         Close (Input);    <--- WARNING: File might be closed
      end;
   end loop;
end Main;
```

- Very simple and targeted abstract interpretation
- DSL to specify new checkers
- Work in progress! Repository here
  `https://github.com/AdaCore/lal-checkers`

Copy paste detector

- Done with Python API too
- Very lightweight (few hundreds lines of code)
- Full article here: `https://blog.adacore.com/a-usable-copy-paste-detector-in-few-lines-of-python`

Applications

- Inside Adacore: change semantic engine in GPS, new versions of GNATmetric, GNATStub, GNATpp
- Outside: clients using it in production for various needs such as:
  - Code instrumentation
  - Automatic refactorings
  - Generation of serializers/deserializers

Conclusion

- Sources are on GitHub: `https://github.com/AdaCore/libadalang`
- Come open issues and create pull requests!
- API is still a moving target