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/alter data about Ada sources
- 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.
- Multi-language: Easy binding generation to other languages/ecosystems
  - Today: Python, Ada, C
- Easy scripting: Be able to create a prototype quickly & interactively
Figure 1: Syntax & block highlighting
The need - IDEs

```
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191

end record;

Type Cond_Branch_Context is limited record
  Decision_Stack : Decision_Occurrence_Vectors.
  -- The stack of open decision occurrences

Basic_Blocks : Basic_Block_Sets.Set;
  -- All basic blocks in the routine being analyzed

Stats : Branch_Statistics;
  -- Statistics on conditional branches in the

Subprg : Address_Info_Acc;
  -- Info of enclosing subprogram
end record;

procedure Analyze_Routine
  (Name : String_Access;
```

Figure 2: Cross references
The need - IDEs

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: type name should start with uppercase letter
main.adb:3:3: variable name should start with lowercase letter
Why not ASIS/GNAT?

Challenges for ASIS’s GNAT implementation

- 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 based ASIS implementation is ill suited to those challenges.

API problems

- ASIS API is too low level/too difficult to change
- Desire for a more modern, higher level API
Why not blank slate implementation of ASIS?

- ASIS specifies a complicated API
- A lot of work to create a new implementation
- And then, it is still not what we want! We still need to:
  - Change most parts of the API.
  - Add a lot of operations (refactoring API, higher level semantic queries, etc..)
  - Specify how error recovery works with ASIS
  - ...

So better to start from scratch :)
API Part 1: Tokens

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

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

Let’s rewrite:

call = unit.root.find(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;
import sys
import libadalang as lal

def check_ident(ident):
    if not ident.text[0].isupper():
        print '{}:{}: variable name "{}" should be capitalized'.format(
            ident.unit.filename, ident.sloc_range.start, ident.text
        )

cxt = lal.AnalysisContext()
for filename in sys.argv[1:]:
    u = cxt.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
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

```python
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,

for b in unit.root.findall(lal.BinOp):
    if interesting_oper(b.f_op) and has_same_operands(b):
        print 'Same operands for {} in {}\n'.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);  
            begin
                Put_Line (Line);
                Close (Input);  
                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
- 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
- First stable version by October 2018
- API will be incrementally improved after that
- We’ll try to avoid breakage as much as possible
- But allow ourselves to make it better for the future :)