Langkit

source code analyzers for the masses

Pierre-Marie de Rodat Raphaël Amiard

Software Engineers at AdaCore

AdaCore

Langkit: A meta compiler

High level vision

A collection of DSLs to implement language parsing and analysis front ends.

Front ends generated by Langkit could be the basis for:

- compilers
- debuggers (e.g. expression evaluator)
- interactive code browsers
- static analyzers
- automatic code refactoring tools

Original use case: Libadalang

- The Ada ecosystem lacks a good library to create code-aware tools
- Several half-backed analyzers in various tools (e.g. GPS, the main IDE)
- Libadalang = Langkit-generated library for Ada-aware tools

The DSL

Syntax

- Python-based DSL for now (will self-host one day!)
- Really several related sub-DSLs: each has its own purpose:
  - express parsing rules
  - describe AST structure
  - turing complete computations
  - ...

DSL Episode 1: Lexing

- Define a list of token kinds:

```
from langkit.lexer import LexerTokenn, WithText, WithSymbol

class MyTokens(LexerToken):
    Def = WithText()
    Identifier = WithSymbol()
    # ...
```

- Provide regexp-based scanning rules to produce them:

```
from langkit.lexer import Lexer, Literal, Pattern

my_lexer = Lexer(MyToken)
my_lexer.add_rules(
    (Literal('def'), MyTokens.Def),
    (Pattern(r'[a-zA-Z][a-zA-Z0-9]*',
     MyTokens.Identifier)),
    # ...
)
```

AdaCore

DSL Episode 2: Tree

- Define lists of AST nodes the parser can produce:

```python
from langkit.dsl import ASTNode, Field

class RootNode(ASTNode):
    pass

class Name(RootNode):
    token_node = True

class Def(RootNode):
    name = Field()

# ...
```

- AST nodes inheritance tree
- Nodes can be abstract
- Optional type annotations on Field

DSL Episode 3: Parsing

- Recursive descent parser combinators (sequences, lists, optional parts, alternatives, ...)
- Packrat parsers
- Add lists of parsing rules
- Specify one default starting one

```
from langkit.parsers import Grammar, List

g = Grammar('main_rule')
g.add_rules(main_rule=List(g.def_rule),
            name=Name(MyTokens.Identifier),
            def_rule=Def('def', g.name),
            # ...
)
```

- Compiling the grammar:
  - infers AST node types Field annotations not present if not present;
  - checks consistency otherwise.

DSL Episode 4: Scoping

- Sub-DSL inside the AST node declarations
- Foundation for semantic analysis
- Create name/AST nodes mappings: lexical environments

```
from langkit.dsl import T
from langkit.envs import EnvSpec, add_env, add_to_env
from langkit.expressions import New, Self

class Function(RootNode):
    local_vars = Field()
    env_spec = EnvSpec(add_env())

class VariableDeclaration(RootNode):
    name = Field()
    env_spec = EnvSpec(add_to_env(mappings=New(
        T.env_assoc, key=Self.name.symbol, val=Self)))
```

DSL Episode 5.1: Semantic analysis

- Sub-DSL inside AST node declarations
- Create methods on AST nodes (called “properties”)
- Public properties: user API for semantic analysis
- Private ones: implementation detail, hidden from users
- Functional programming language

```
from langkit.expressions import langkit_property

class VariableReference(FooNode):
    name = Field()

    @langkit_property(public=True)
    def var_decl():
        return Self.node_env.get_first(Self.name)
```

DSL Episode 5.2: Logic DSL

- Sub-DSL inside the properties DSL
- Several expressions to create logic equations
- Equation solutions give semantic analysis' output

DSL Episode 5.3: Logic DSL example (1/n)

```
def f(a)
def f(a, b)

f(1)
f(2, 3)
f(2, 3, 4)
```

DSL Episode 5.3: Logic DSL example (2/n)

```python
class Function(FooNode):
    name = Field()
    arguments = Field()

    env_spec = EnvSpec(add_to_env(
        mappings=New(T.env_assoc, key=Self.name.symbol, val=Self)
    ))

    @langkit_property()
    def args_match(call_args=T.IntegerLiteral.list):
        return call_args.length == Self.arguments.length
```

DSL Episode 5.4: Logic DSL example (3/n)

```
class Call(FooNode):
    name = Field()
    arguments = Field()

    called_var = UserField(type=T.LogicVarType, public=False)

    @langkit_property(public=True)
    def resolve():
        return Entity.equation.solve

    @langkit_property()
    def equation():
        candidates = Var(Entity.node_env.get(Self.name).filtermap(
            lambda c: c.el.cast(Function),
            lambda c: c.is_a(Function)
        ))
        return candidates.logic_any(lambda f: Entity.sub_equation(f))

    @langkit_property()
    def sub_equation(func=Function):
        return (Bind(Entity.called_var, func)
                & Predicate(T.Function.args_match,
                            Entity.called_var,
                            Self.arguments))

    @langkit_property(public=True)
```

Ad

DSL Episode 5.5: Why?

- Previous example would be simpler without equations
- Just filter the list of candidates and return the first one, right?
- Main use case: overload resolution

```
type Integer;
type Character;
type Float;

function F1 (I : Integer) return Integer;
function F1 (F : Float) return Integer;

function F2 (C : Character) return Integer;
function F2 (C : Character) return Character;

F1 (1);
F1 ('C');
F2 (2);

F1 (F2 (F2 ('C')));
```

Crafted for incremental analysis

- Reloading happens a lot in IDE: performance required
- Avoid big recomputations for common operations
- No need to recompute *everything* when reloading one source file:
- Keep source file-specific data as much isolated as possible
- Reduced update process when removing/reloading source files

The generated libraries

Requirements for the target language:

- Fast
- Low level enough
- Memory management agnostic (no GC)
- Easy to bind to C and other languages

Candidates

- C, C++, Ada, Rust, ...

Chosen one: Ada

- Since the project is developed at AdaCore: no surprises :)

Bindings to other languages

Automatically generated C bindings

So that it is very easy to generate bindings to any languages the users wants.

First class citizen Python bindings

- Python is the de-facto scripting language of the Langkit ecosystem.
- Everything possible in Ada is possible in Python

Use multiple generated libraries from python !

```
import libadalang as lal # Langkit generated lib for Ada
import libpythonlang as lpl # Langkit generated lib for Python

ada_ctx = lal.AnalysisContext()
python_ctx = lpl.AnalysisContext()

print ada_ctx.get_from_buffer("<buffer>", """
procedure Main is
begin
   null;
end Main;
"""
) # <CompilationUnit 2:1-5:10>
print python_ctx.get_from_buffer("<buffer>", """
def test(a, b):
    return a + b
"""
) # <FileNode 1:1-4:1>
```

AdaCore

Easy to generate bindings to new languages

- No need for external bindings generators
- Knowledge about data types, functions, memory management -> Langkit
- Planned in the future:
  - Java (certainly) for interaction with IDEs
  - Lua (maybe)
  - … Whatever you need !

Tree walking

Source code:

```
a = 12
b = 15
print a + b
```

Processing:

```
>>> for assign in unit.root.findall(lpl.AssignStmt):
>>>    print "Stmt: ", assign.text, assign.sloc

Stmt:  a = 12 2:1-2:7
Stmt:  b = 15 3:1-3:7
```

Rewriting

Source code:

```
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()
```

Result:

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

Generic tools shipping with the libraries

AdaCore

Small tools

./playground

- Command line tool based on IPython
- Allow interactive exploration of the tree/API in general

./parse

- Allows you to inspect the structure of the tree
- Dump lexical environments

Unparser (along with tree rewriting)

- Create a new source file *only from the tree* (not using original source information)
- Can also be used to create sources from completely synthetic trees
- Uses the grammar and the AST definition (no additional code needed)

Code indenter (prototype)

- Provide a declarative data structure for indentation rules

```
block_rule = field_rules(constant_increment=3)
paren_rule = field_rules(on_token="(")

indent_map = {
    lal.PackageDecl: Indent(
        field_rules=indent_fields(
            public_part=block_rule, private_part=block_rule
        )
    ),
    ...
    lal.Params: Indent(
        field_rules=indent_fields(params=paren_rule)
    ),
}
```

- Get auto indentation on tab in your favorite editor

Syntax highlighter (not done)

- Auto generation of syntax highlighter
- Highlight keywords by default
- Custom rules to highlight more complex syntax based rules
- Automatic support in your editor

Language server protocol? (not done)

- Tentative plan: Automatically generate basic LSP support from the plug-in
- We have a Neovim plug-in already doing for Ada:
  - Indentation
  - Go to definition
  - Tree editing and exploration
- In the future: More editors, more languages ?

A multi-language static analyzer in 20 lines of Python

```python
ada_ops = (lal.OpMult, lal.OpPlus, lal.OpDoubleDot, lal.OpPow, lal.OpConcat)
py_ops = (pyl.OpMult, pyl.OpPlus, pyl.OpDoubleDot, pyl.OpPow, pyl.OpConcat)

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)

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)
```

AdaCore

Existing langkit-based libraries & prototypes

- Ada
- Python
- JSON
- GPR files (AdaCore's project description language)
- KConfig