Pyrser Selectors Language

Lionel Auroux - LSE Summer Week - 2015

Summary

- What?
- So we need...
- Tree Automata.
- PSL Basic Syntax.
- PSL Algorithm.
- PSL Advanced Syntax.
- Conclusion.

What?

Classical Compilation Scheme:

- Parsing -> AST Creation.
- ! Validating & AST Desugaring -> AST visiting.
- ! Type Checking -> AST visiting.
- ! Generation -> AST visiting.

What?

sample python

```
>>> class v(ast.NodeVisitor):
...   def generic_visit(self, node):
...     print(type(node).__name__)
...     ast.NodeVisitor.generic_visit(self, node)
>>> x = v()
>>> t = ast.parse('d[x] += v[y, x]')
>>> x.visit(t)
```

What?

sample python

```
>>> class allnames(ast.NodeVisitor):
...   def visit_Module(self, node):
...     self.names = set()  # manage a state thru the
visitor
...     self.generic_visit(node)
...     print sorted(self.names)
...   def visit_Name(self, node):
...     self.names.add(node.id)
>>> x = v()
>>> t = ast.parse('d[x] += v[y, x]')
>>> x.visit(t)
```

What?

By AST visiting:

- we visit only some kinds of node by pass
- we handle the states between different kinds of node

Semantic = many passes, many states shared

What?

AST Visiting a common way to handle these steps, but:

- Language semantic -> A mess of handwritten code

➔ Hard to understand, Hard to maintain

So we need...

Before:

- many methods and states on many files/passes

After:

- one automated method to understand what happen

➔ A centralized DSL to describe what to match/transform

So we need...

We could notice, that’s for simple match/transform:
AST visiting is easy to write

A Specialized DSL is a bit too much

It’s a uniform and generic way to do things...

Tree Automata

A way to walk thru a tree and match some construction.

Some techno using tree automata:

- XPath:
  - `B/*[1]` : first child of B whatever its name
- CSS Selector:
  - `div > p` : all <p> with as parent

Tree Automata

But in CSS Selector or XPath library:

The main function takes only one pattern and collects all elements in tree corresponding to that.

For tree transformation with our DSL, we need to take many patterns that match in parallel (multi matching).

Tree Automata

In theory: Tree Automata Techniques & Application, October, 12th 2007 (aka Tata). Algorithm classification, proofs…

Mainly 2 kind of algorithms:

- Top-Down: root to leaf matching/walking rules
- Bottom-Up: leaf to root…

Deterministic or not:

2 sames rules in LHS -> nondeterministic.

Tree Automata

But in practice: no good implementation…

Xpath, CSS Selector:

- Top-down automata.
- Collecting, no parallel (multi) matching.

Tree Automata

In CSS Sel/Xpath, the pattern language denotes the Top Down algorithms.

We need to write our pattern in natural order but transform it into a Bottom Up tree automata.

PSL Basic Syntax

So we design our DSL for the pyrser toolbox.

Named PSL aka Pyrser Selectors Language due to the CSS Selector inspiration for pattern language.

Match all kind of python object's tree (Pyrser's AST or not).

PSL Basic Syntax

examples:

```
{
     TypeNode1() -> #hookPython;
     TypeNode2(.attributes=12) -> #hookPython;
     TypeNode3([2: “toto”]) -> #hookPython;
     TypeNode4({‘key’: 1.2 }) -> #hookPython;
}
```

Here `TypeNode*` denotes python types, `.attributes` denotes attributes names. `[ ]` list, `{}` dict

PSL Basic Syntax

We could describe strictly an object by using all:

```
Node(.a=12, .c=”toto”, .d=list([0: 12])) -> #hookPython ;
```

Note: Here we are describing EXACTLY the node…

We need wildcards, and unstrict matching...

PSL Basic Syntax

#hookPython is a python function with a certain prototype.

```
def hookPython(tree: Node, user_data, nodes: list)
```

`tree`: the current node

`user_data`: a data pass to the matching function

`nodes`: a list of all previous matched types

- with just one pattern `nodes[0]` alias of tree
- `nodes[-1]` the parent node

PSL Basic Syntax

```
def hookPython(tree: Node, user_data, nodes: list):
    #… if tree is in an attributes
    nodes[-1] = NewNode(tree)
    #...if tree is in a list
    index = nodes[-1].index(tree)
    nodes[-1][index] = NewNode(tree)
    #...if tree is in a dict
    key=nodes[-1].keys()[nodes[-1].values().index(tree)]
    nodes[-1][key] = NewNode(tree)
```

PSL Algorithm

DFS (Depth First Search)

<on return>

Automata

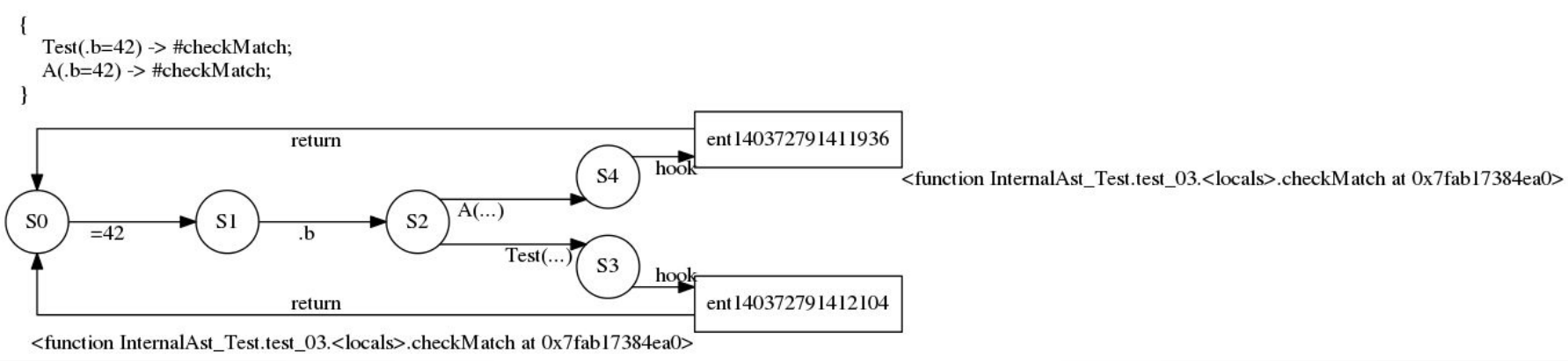
States {Live, Events, Die}

PSL Algorithm

How to transform this basic syntax in tree automata:

1. The LHS represent what we want but in TD order
2. For each statement, we walk thru LHS by another DFS and construct our automata rules in reverse order.
3. We found and merge all common base *
4. At worst case, each statement provide a tree automata for multi-matching

PSL Algorithm



S0 is the initial state, if the arrow was not taken we return to S0

PSL Algorithm

When merge fail, the statement will be match in another instance (none deterministic: same LHS different RHS).

Due to multi-match

- ➔ We have many instance referring to this automata
- ➔ These instances are living or died during walking
- ➔ Each instances could share a specific context

So we could set “events” during matching to condition some expression.

PSL Advanced Syntax

examples of event use:

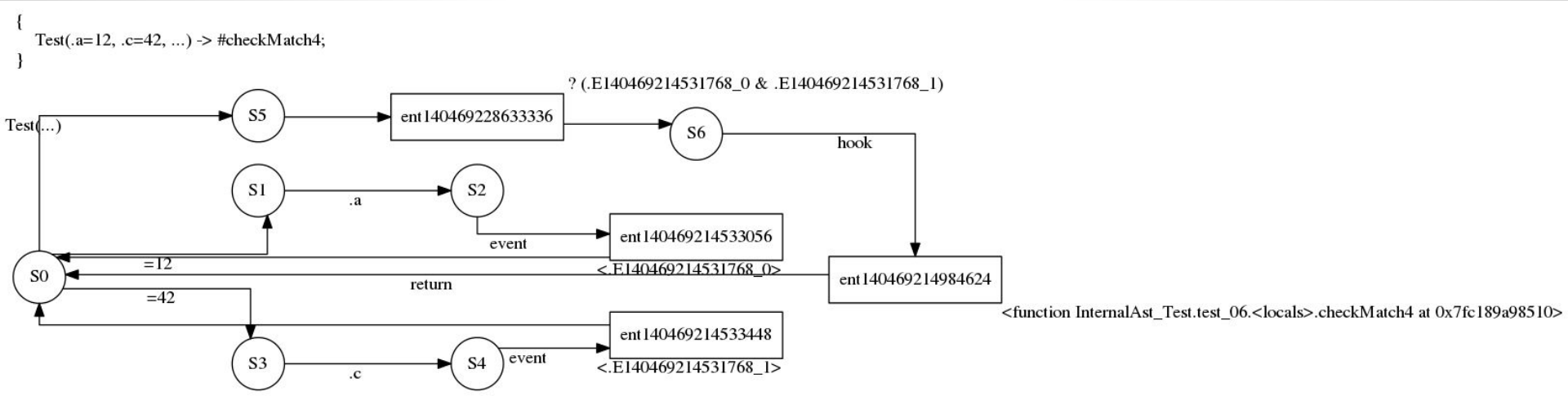
```
{
     Default() -> <DEFAULT>;
     Case() -> <CASE>;
     Switch() ? (CASE|DEFAULT>  -> #ok;
     Func() ? <CASE|DEFAULT> -> #notok;
}
```

when the event was checked by ? expression, we set it off.

PSL Advanced Syntax

So now we could create non strict match.

```
Test(.a=12, .c=42, ...) -> #checkMatch;
```



PSL Advanced Syntax

We could extended our language to support some ambiguous constructions:

- * wildcard for unspecified name,value, indice, key

```
{
  T(.*= 12, .c=*, [*: “toto”], {*: 1.2 }…)
   -> #doIt;
}
```

PSL Advanced Syntax

Events could be checked without be set it off:

- ?? to add a precondition, events are not set it off
- ? to add precondition, events are set it off

```
{
   A(...) -> <child>;
   B(...) ?? (child) -> <child2>
   C(...) ? (child & child2) -> #doIt;
}
```

PSL Advanced Syntax

As we just seen, we could chained Nodes by ancestor's relationship.

```
{
    C(...) > B(...) > A(...) -> #doIt;
}
```

Here
```
nodes[0] -> C()
nodes[1] -> B()
nodes[2] -> A()
```

PSL Advanced Syntax

We could match “IsKindOfType”

```
{
   Class(...) -> #hook1;
   Base^(...) -> #hook2;
}
```

Non-determinism: If Class inherit from Base, then hook1 & hook2 are executed in declaration order.

Conclusion

Todo:

- Add some operator of CSS Selector for value matching (^= leftString, $= rightString, …)
- Handle set data type and set value
- !Type (root node without Type in path)
- Siblings
- Uses cases with Cnorm

?