Identifier Binding Computation

Date Mon 18 April 2016 **By** serge-sans-paille **Category** compilation.

Foreword

This is **not** a Jupyter notebook, but it could have been. Instead, the content of this article is mostly taken from the Doctest of the `pythran.analyses.aliases` module, and the relevant unit tests in `test_typing.py`. So the reader still has a strong warranty that the output described is the one she would get by running the commands herself.

The curious reader can verify this statement by running `python -m doctest` with Pythran in its `PYTHONPATH` [git-version] on the article source, which is in fact what I did before posting it `:-)`.

Static Computation of Identifier Binding

In Python, everything is a reference, from literal to objects. Assignment creates a *binding* between a reference and an identifier, thus the following sequence always hold:

```
>>> a = list()
>>> b = c = a
>>> d = b
>>> d is a
True
```

In some sense, assignement creates aliasing between identifiers, as any change made to the *value* referenced by the identifier `b` impacts the *value* referenced by identifer `c` (and `a` and `d`):

```
>>> a.append(1)
>>> len(b) == len(c) == len(d) == len(a) == 1
True
```

In the context of Pythran, the static knowledge of the different values of an identifier **may** be bound to, is critical. First there is no reason to trust an identifier, as shown by the following code:

```
>>> id = len
>>> id([1])
1
```

Nothing prevents this to happen in Python [0], so Pythran takes great care in not confusing *identifiers* and *values*. And The ill-named Alias Analysis is the tool we use to solve this problem. In the particular case above, this analysis tells us that the identifier `id` in the call expression `id([1])` always has the value `__builtin__.len`. This can be used, for instance, to state that this call has no side effect.

Where is Identifier Binding Used in Pythran

Identifier binding is used by all Pythran analyses that interact with function calls, when they need to know something about the function property, or when they want to verify that all the possibles (function) values taken by an identifier share the same property. For instance:

> 1. Conversion from calls with named arguments to call without named arguments, as in `zeros(10, dtype=int)`
> 2. Conversion from iterator to generator, e.g. turning `range` into `xrange` (Python2 inside `:-/`)
> 3. Constant folding (it needs to make sure it manipulates pure functions)
> 4. …

But the single more important use of identifier binding is in fact, typing. This is likely to evolve, but current (clumsy) typing system in Pythran attaches some kind of typing properties to functions. For instance for the following function:

```
>>> def foo(x, y): x.append(y)
```

Pythran computes a property that states

> > if functions `foo` is called with an argument of type A as first argument and an argument of type B as second argument, > **then** the type of the first argument is the combination of its actual type A and an abstract type *Container of B*

So in case we make the following call:

```
>>> a = b = []
>>> foo(a, 1)
```

then the type of `a` is first computed to be *empty list* and calling `foo` combines this information with the fact that `a` must be capable of holding integers, to conclude `a` has the type *list of integers*.

Identifier binding is used twice in the process. Once to prove that the *identifier* `foo` is bound to the value `foo`, and once to track which values the *identifier* `a` was bound to; here to compute that the type information gathered for `a` also impacts `b`, even if `b` was not used in the function call, as they share the same value.

Computing an Overset of the Bound Values

Pythran **cannot** track any possible values bound to a variable. In the following example:

```
>>> for i in range(1000):
```

```
...     pass
```

identifier `i` can be bound to a great deal of values, and we cannot track them individually. Instead Pythran only keep tracks of values that are bound to an identifier. All the others are hidden between the terms of `<unbound-value>`.

So let's start to write some simple equations [1], with a few test cases demonstrated as Python code which needs some initialization:

```
>>> import ast
>>> from pythran.analyses.aliases import *
>>> from pythran import passmanager
>>> pm = passmanager.PassManager('demo')
```

Here, we basically inject the `aliases` namespace into current namespace for convenience, then create an instance of the object in charge of applying passes and gathering analysis results.

Bool Op Expression

(A.k.a ``or`` and ``and``)

Resulting node may alias to either operands:

```
>>> module = ast.parse('def foo(a, b): return a or b')
>>> result = pm.gather(Aliases, module)
>>> Aliases.dump(result, filter=ast.BoolOp)
(a or b) => ['a', 'b']
```

This code snippet requires a few explanations:

1. First, it parses a code snippet and turns it into an Abstract Syntax Tree (AST).
2. **Second, it computes the alias information at every point of the program.**
   `result` is a dictionary that maps nodes from the AST to set of identifiers (remember that for Pythran, a node can only alias to bounded values. These values are represented by the first identifier they are bound to).

3. **Finally, it pretty prints the result of the analysis, using a filter to**
   only dump the part we are interested in. In that case it dumps a textual representation of the alias set of the `ast.BoolOp` nodes, which turns out to be `['a', 'b']`.

Unary Operator Expression

Resulting node does not alias to anything

```
>>> module = ast.parse('def foo(a): return -a')
>>> result = pm.gather(Aliases, module)
>>> Aliases.dump(result, filter=ast.UnaryOp)
(- a) => ['<unbound-value>']
```

As stated previously, values not bound to an identifier are only represented as `<unbound-value>`.

If Expression

Resulting node alias to either branch

```
>>> module = ast.parse('def foo(a, b, c): return a if c else b')
>>> result = pm.gather(Aliases, module)
>>> Aliases.dump(result, filter=ast.IfExp)
(a if c else b) => ['a', 'b']
```

Dict Expression

A dict is abstracted as an unordered container of its values

```
>>> module = ast.parse('def foo(a, b): return {0: a, 1: b}')
>>> result = pm.gather(Aliases, module)
>>> Aliases.dump(result, filter=ast.Dict)
{0: a, 1: b} => ['|a|', '|b|']
```

where the `|id|` notation means something that may contain `id`.

Set Expression

A set is abstracted as an unordered container of its elements

```
>>> module = ast.parse('def foo(a, b): return {a, b}')
>>> result = pm.gather(Aliases, module)
>>> Aliases.dump(result, filter=ast.Set)
{a, b} => ['|a|', '|b|']
```

Tuple Expression

A tuple is abstracted as an ordered container of its values

```
>>> module = ast.parse('def foo(a, b): return a, b')
>>> result = pm.gather(Aliases, module)
>>> Aliases.dump(result, filter=ast.Tuple)
(a, b) => ['|[0]=a|', '|[1]=b|']
```

where the `|[i]=id|` notation means something that may contain `id` at index `i`.

Call Expression

Resulting node alias to the return_alias of called function, if the function is already known by Pythran (i.e. it's an Intrinsic) or if Pythran already computed it's `return_alias` behavior.

```
>>> fun = '''
... def f(a): return a
... def foo(b): c = f(b)'''
>>> module = ast.parse(fun)
```

The `f` function create aliasing between the returned value and its first argument.

```
>>> result = pm.gather(Aliases, module)
>>> Aliases.dump(result, filter=ast.Call)
f(b) => ['b']
```

This also works with intrinsics, e.g. `dict.setdefault` which may create alias between its third argument and the return value.

```
>>> fun = 'def foo(a, d): __builtin__.dict.setdefault(d, 0, a)'
>>> module = ast.parse(fun)
>>> result = pm.gather(Aliases, module)
>>> Aliases.dump(result, filter=ast.Call)
__builtin__.dict.setdefault(d, 0, a) => ['<unbound-value>', 'a']
```

Note that complex cases can arise, when one of the formal parameter is already known to alias to various values:

```
>>> fun = '''
... def f(a, b): return a and b
... def foo(A, B, C, D): return f(A or B, C or D)'''
>>> module = ast.parse(fun)
>>> result = pm.gather(Aliases, module)
>>> Aliases.dump(result, filter=ast.Call)
f((A or B), (C or D)) => ['A', 'B', 'C', 'D']
```

Subscript Expression

The resulting node alias only stores the subscript relationship if we don't know anything about the subscripted node.

```
>>> module = ast.parse('def foo(a): return a[0]')
>>> result = pm.gather(Aliases, module)
>>> Aliases.dump(result, filter=ast.Subscript)
a[0] => ['a[0]']
```

If we know something about the container, e.g. in case of a list, we can use this information to get more accurate informations:

```
>>> module = ast.parse('def foo(a, b, c): return [a, b][c]')
>>> result = pm.gather(Aliases, module)
>>> Aliases.dump(result, filter=ast.Subscript)
```

```
[a, b][c] => ['a', 'b']
```

Moreover, in case of a tuple indexed by a constant value, we can further refine the aliasing information:

```
>>> fun = '''
... def f(a, b): return a, b
... def foo(a, b): return f(a, b)[0]'''
>>> module = ast.parse(fun)
>>> result = pm.gather(Aliases, module)
>>> Aliases.dump(result, filter=ast.Subscript)
f(a, b)[0] => ['a']
```

Nothing is done for slices, even if the indices are known :-/

> ```
> >>> module = ast.parse('def foo(a, b, c): return [a, b, c][1:]')
> >>> result = pm.gather(Aliases, module)
> >>> Aliases.dump(result, filter=ast.Subscript)
> [a, b, c][1:] => ['<unbound-value>']
> ```

List Comprehension

A comprehension is not abstracted in any way

```
>>> module = ast.parse('def foo(a, b): return [a for i in b]')
>>> result = pm.gather(Aliases, module)
>>> Aliases.dump(result, filter=ast.ListComp)
[a for i in b] => ['<unbound-value>']
```

Return Statement

A side effect of computing aliases on a Return is that it updates the `return_alias` field of current function

```
>>> module = ast.parse('def foo(a, b): return a')
>>> result = pm.gather(Aliases, module)
>>> module.body[0].return_alias # doctest: +ELLIPSIS
<function merge_return_aliases at...>
```

This field is a function that takes as many nodes as the function argument count as input and returns an expression based on these arguments if the function happens to create aliasing between its input and output. In our case:

```
>>> f = module.body[0].return_alias
>>> Aliases.dump(f([ast.Name('A', ast.Load()), ast.Num(1)]))
['A']
```

This also works if the relationship between input and output is more complex:

```
>>> module = ast.parse('def foo(a, b): return a or b[0]')
>>> result = pm.gather(Aliases, module)
>>> f = module.body[0].return_alias
>>> List = ast.List([ast.Name('L0', ast.Load())], ast.Load())
>>> Aliases.dump(f([ast.Name('B', ast.Load()), List]))
['B', '[L0][0]']
```

Which actually means that when called with two arguments B and the single-element list `[L[0]]`, `foo` may returns either the first argument, or the first element of the second argument.

Assign Statement

Assignment creates aliasing between lhs and rhs

```
>>> module = ast.parse('def foo(a): c = a ; d = e = c ; {c, d, e}')
>>> result = pm.gather(Aliases, module)
>>> Aliases.dump(result, filter=ast.Set)
{c, d, e} => ['|a|', '|a|', '|a|']
```

Everyone points to the formal parameter a o/

For Statement

For loop creates aliasing between the target and the content of the iterator

```
>>> module = ast.parse('''
... def foo(a):
...     for i in a:
...         {i}''')
>>> result = pm.gather(Aliases, module)
>>> Aliases.dump(result, filter=ast.Set)
{i} => ['|i|']
```

Not very useful, unless we know something about the iterated container

```
>>> module = ast.parse('''
... def foo(a, b):
...     for i in [a, b]:
...         {i}''')
>>> result = pm.gather(Aliases, module)
>>> Aliases.dump(result, filter=ast.Set)
{i} => ['|a|', '|b|']
```

If Statement

After an if statement, the values from both branches are merged, potentially creating more aliasing:

```
>>> fun = '''
```

```
... def foo(a, b):
...     if a: c=a
...     else: c=b
...     return {c}'''
>>> module = ast.parse(fun)
>>> result = pm.gather(Aliases, module)
>>> Aliases.dump(result, filter=ast.Set)
{c} => ['|a|', '|b|']
```

Illustration: Typing

Thanks to the above analysis, Pythran is capable of computing some rather difficult informations! In the following:

```
def typing_aliasing_and_variable_subscript_combiner(i):
    a=[list.append,
       lambda x,y: x.extend([y])
    ]
    b = []
    a[i](b, i)
    return b
```

Pythran knows that b is a list of elements of the same type as `i`.

And in the following:

```
def typing_and_function_dict(a):
    funcs = {
        'zero' : lambda x: x.add(0),
        'one' : lambda x: x.add(1),
    }
    s = set()
    funcs[a](s)
    return s
```

Pythran knows that s is a set of integers :-)

Illustration: Dead Code Elimination

Consider the following sequence:

```
>>> fun = '''
... def useless0(x): return x + 1
... def useless1(x): return x - 1
... def useful(i):
...     funcs = useless0, useless1
...     funcs[i%2](i)
...     return i'''
```

Pythran can prove that both `useless0` and `useless1` don't have side effects. Thanks to the binded value analysis, it can also prove that **whatever** the index, `funcs[something]` either points to `useless0` or `useless1`. And in either cases, the function has no side effect, which means we can remove the whole instruction:

```
>>> from pythran.optimizations import DeadCodeElimination
>>> from pythran.backend import Python
>>> module = ast.parse(fun)
>>> _, module = pm.apply(DeadCodeElimination, module)
>>> print pm.dump(Python, module)
def useless0(x):
    return (x + 1)
def useless1(x):
    return (x - 1)
def useful(i):
    funcs = (useless0, useless1)
    pass
    return i
```

Other optimizations will take care of removing the useless assignment to `funcs` :-)

Acknowledgments

Thanks a lot to Pierrick Brunet for his careful review, and to Florent Cayré from Logilab for his advices that helped **a lot** to improve the post. And of course to OpenDreamKit for funding this work!

[0] Except the sanity of the developer, but who never used the `id` or `len` identifiers?

[1] Starting from this note, the identifiers from the ast module are used.

[git-version] The Pythran commit id used for this article is `f38a16491ea644fbaed15e8facbcabf869637b39`

TAGS