REPORT ON OpenDreamKit DELIVERABLE D5.4

Make PYTHRAN typing better to improve error information.

SERGE GUELTON

Due on	31/08/2016 (Month 12)
Delivered on	02/10/2017
Lead	Logilab (Logilab)
Progress on and finalization of this deliverable has been tracked publicly at: `https://github.com/OpenDreamKit/OpenDreamKit/issues/117`	

DELIVERABLE DESCRIPTION, AS TAKEN FROM GITHUB ISSUE #117 ON 2017-02-10

- **WP5:** High Performance Mathematical Computing
- **Lead Institution:** Logilab
- **Due:** 2016-08-31 (month 12)
- **Nature:** Demonstrator
- **Task:**
- **Proposal:** p.51
- **Final report**

Pythran is a Python to C++ compiler for a subset of the Python language, with a focus on scientific computing, which takes advantage of multi-cores and SIMD instruction units. Given the importance of Python in the OpenDreamKit ecosystem, Pythran is one of the promising building blocks for high performance mathematical computing.

This deliverable is about enhancing the Pythran compiler to provide better user feedback when a type error is meet, by extracting and taking advantage of fine grain type information. The optimizations developed in this context are a more accurate version of identifier binding that increases the scope of Pythran valid input. Second, an unsound type checker has been developed and integrated in the 0.8 version of the Pythran compiler. It provides usable, meaningful feedback to users in case of type error, instead of internal compiler errors. These two steps make it easier to write computation-intensive kernels to be compiled by Pythran.

CONTENTS

Appendix A. 18 Apr 2016, Identifier Binding Computation, Pythran blog

http://serge-sans-paille.github.io/pythran-stories/identifier-binding-computation.html

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`

BLOGROLL

Pythran Doc

Pythran on PyPI

SOCIAL

atom feed

github

CATEGORIES

benchmark

compilation

examples

optimisation

release

Appendix B. 10 Dec 2016, from pythran import typing, Pythran blog

http://serge-sans-paille.github.io/pythran-stories/from-pythran-import-typing.html

from pythran import typing

Date Sat 10 December 2016 By serge-sans-paille Category compilation.

Pythran is currently part of OpenDreamKit, a project that aims at improving the open source computational mathematics ecosystem.

The goal of Pythran is indeed to improve some Python kernel computations, but there's something that actually makes Pythran difficult to use for new comers. What is it? Let's have a look at the following Python code, inspired by a stack overflow thread:

```
#pythran export create_grid(float [])

import numpy as np
def create_grid(x):
    N = x.shape[0]
    z = np.zeros((N, N))
    z[:,:,0] = x.reshape(-1,1)
    z[:,:,1] = x
    fast_grid = z.reshape(N*N, 3)
    return fast_grid
```

An attempt to compile it with Pythran would return a very long C++ template instantiation trace, with very little clue concerning the origin of the problem.

```
> pythran create_grid.py
In file included from /tmp/tmpP0xYa2.cpp:10:
In file included from ./pythran/pythonic/include/types/ndarray.hpp:12:
In file included from ./pythran/pythonic/include/utils/broadcast_copy.h
pp:4:
./pythran/pythonic/include/types/tuple.hpp:122:25: error: array is too
large (18446744073709551615 elements)
      value_type buffer[N ? N : 1];
                        ^~~~~~~~~
./pythran/pythonic/include/types/numpy_iexpr.hpp:57:26: note: in instan
tiation of template class 'pythonic::types::array<long, 184467440737095
51615>' requested here
      array<long, value> _shape;
                         ^
[...]
                                                   ^
./pythran/pythonic/include/utils/seq.hpp:19:19: note: use -ftemplate-de
```

```
pth=N to increase recursive template instantiation depth
    struct gens : gens<N - 1, N - 1, S...> {
                  ^
3 errors generated.
CRITICAL Cover me Jack. Jack? Jaaaaack!!!!
E: ('error: Command "clang++-3.8 -DNDEBUG -g -fwrapv -O2 -Wall -fno-str
ict-aliasing -g -O2 -fPIC -DUSE_GMP -DENABLE_PYTHON_MODULE -D__PYTHRAN_
_=2 -I./pythran -I./pythran/pythonic/patch -I/home/serge/.venvs/pythran
/local/lib/python2.7/site-packages/numpy/core/include -I/usr/include/py
thon2.7 -c /tmp/tmpP0xYa2.cpp -o /tmp/tmpM2Eiso/tmp/tmpP0xYa2.o -std=c+
+11 -fno-math-errno -w -fwhole-program -fvisibility=hidden" failed with
 exit status 1',)

What we now have is a slightly friendlier message:
```

```
> pythran create_grid.py
CRITICAL You shall not pass!
E: Dimension mismatch when slicing `Array[2d, float]` (create_grid.py,
line 7)
```

Indeed, the correct declaration for z was `z = np.zeros((N, N, 3))`.

A Quick Glance at Pythran Typing System

As you probably know, Python uses a dynamic type system, often called *duck typing*: what matters is not the type of an object, but its structure, i.e. the available methods and fields: *If it walks like a duck and talks like a duck, then it's a duck*. That's a kind of structural typing.

On the opposite side C++ uses a static type system, and if you adhere to OOP [1] you may require an object to derive from the `Duck` class to be considered a duck; That's a kind of nominal typing.

Pythran uses a trick to make both world meet: *ad-hoc polymorphism*, as supported in C++ through `template` meta programing. Upon a template instantiation, there's no type name verification, only a check that given methods and attributes make sense in the current context. And that's exactly what we need to get closer to Python typing!

This all is very nice, except in the case of a bad typing. Consider this trivial Python code:

```
def twice(s):
    return s * 2
```

integer, for instance `str`, `list`, `int`. The C++ equivalent would be (taking into account move semantics):

```
template<typename T>
auto twice(T&& s) {
    return std::forward<T>(s) * 2;
}
```

In Python's case, type checking is done at runtime, during a lookup in `s` for a `__mul__` magic method. In C++ it's done at compile time, when performing instantiation of `twice` for a given type value of T. What lacked was a human-readable error message to warn about the coming winter. And that's exactly the topic of this post ;-)

A Few Words About MyPy

Type hints, as introduced by PEP484, make it possible to leverage on arbitrary function annotations introduced by PEP 3107 to specify the expected type of a function parameter and its resulting return type. No check occur at runtime, but a third party compiler, say MyPy can take advantage of these hints to perform an ahead-of-time check. And that's **great**.

Note

In this post, we use the type annotation introduced by PEP484 and used in MyPy to describe types. `int` is an integer, `List[str]` is a list of string and so on.

So, did we trade `#pythran export twice(str)` for `def twice(s: str):`? No. Did we consider the option? Yes. First there's the issue of MyPy only running on Python3. It can process Python2 code, but it runs on Python3. We've been struggling so much to keep Python2.7 compatibility in addition to the recent addition of broader Python3 support. We're not going to leave it apart without good reasons.

Note

It also turns out that the `typing` module has a different internal API between Python2 and Python3. This makes it quite difficult to use for my purpose. What a joy to discover this when you think you're done with all your tests :-/

No, the main problem is this MyPy issue that basically states that Numpy does not fit into the model:

> Of course, the best behavior would be to provide a stub for Numpy, but some features in Numpy make it difficult to provide a good stub

Meanwhile, someone that did not read this issue wrote A Numpy stub for MyPy. It turns out that it' **is** a pain, mostly due to the flexibility of many Numpy methods.

Additionally, Pythran currently infers type inter-procedurally, while MyPy requires type annotation on every functions, to keep the problem within reasonable bounds.

But wait. MyPy author did his PhD on the subject, and he now works hand in hand with Guildo van Rossum on the subject. Is there any chance for us to do a better job? Let's be honest. There is not.

What can we do in such a situation? Take advantage of some extra assumptions Pythran can afford. We focus on scientific computing, all existing types are known (no user-defined types in Pythran) and we only need to handle small size kernels, so we can spend some extra computing resources in the process.

A Variant of Hindley-Milner for Pythran

Hindley-Milner (HM) is a relatively easy to understand type system that supports parametric polymorphism. A simple implementation has been written in Python, but *not* for Python, even not for the subset supported by Pythran.

The main issue comes with overloaded functions. Consider the `map` function: it has a varying number of parameters, and for a given number of parameters, two possible overloads exist (the first argument being `None` or a `Callable`). Some extra stuff are not as critical but also important: it's not possible to infer implicit option types (the one that comes with usage of `None`). Ocaml uses `Some` as a counterpart of `None` to handle this issue. but there's no such hint in Python (and we don't want to introduce one).

Still, the whole subject of typing is reaaaaaallllly difficult, and I wanted to stick as close as possible to Hindley-Milner because of its simplicity. So what got introduced is the concept of `MultiType`, which is the type of an object that can hold several types at the same time. So that's not exactly a `UnionType` which is the type of an object that can be of one type among many. The difference exists because of the situation described by the following code:

```
def foo(l, m=1):
    pass

foo(1)
foo(2, 3)
```

In that case foo really has two types, namely `Callable[[Any], None]` and `Callalble[[Any, Any], None]`. That's what `MultiType` represents.

Handling Overloading

So we handle overloading through a unique object that has a specific type, a `MultiType` that is just a list of possible types.

Abusing from `Multiype` can quickly make the combinatorics of the type possibilities go wild, so we had to make a decision. Consider the following code:

```
def foo(x, y):
    return x in y
```

The `in` operator could be implemented as a `MultiType`, enumerating the possible valid signature (remember we know of all possible types in Pythran):

- `Callable[[List[T0], T0], bool]`, a function that takes a list of `T0` and a `T0` and returns a boolean,
- `Callable[[str, str], bool]`, a function that takes two strings and returns a boolean,

And so on, including for numpy arrays, but we'll comme back to this later and assume for now we only have these two types. So what is the type of `foo`? From the `x in y` expression, HM tells us that `x` can be a list of `T0`, and in that case `y` must be of type `T0`, **or** `x` is a string and so must be `y`. And in both cases, a boolean is returned.

We could consider both alternatives, follow the two type paths and in the end, compute the signature of `foo` as a `MultiType` holding the outcome of all paths. But that could mean a lot! What we do is an over-approximation: what is the common structure between `List[T0]` and `str`? Both are iterable, therefeore `x` must be iterable. Nothing good comes from `T0` and `str`, and `bool` compared to `bool` results in a `bool`, so in the end `foo` takes an iterable and any value, and returns a boolean. That's not as strict as it could be, but that's definitively enough. However our type system is no longer *sound* (it does not reject all bad program).

In order to make it easier to perform this approximation, we chose a dedicated representation for containers. In our type system (oh, it's named *tog* by the way, so in the tog type system), containers are roughly described as a tuple of `(name, sized, key, value, iter)`:

- a `List[T0]` is considered as `(List, Sized, int, T0, T0)`
- a `Set[T0]` is considered as `(Set, Sized, NoKey, T0, T0)`
- a `Dict[T0, T1]` is considered as `(Dict, Sized, T0, T1, T0)`
- a `str` is considered as `(Str, Sized, int, Str, Str)`
- a `Generator[T0]` is considered as `(Generator, NoSized, NoKey, T0, T0)`

As a consequence, an `Iterable[T0]`, to be compatible with the over-approximation defined above, is a `(Any, Any, Any, Any, T0)`.

Handling Option Types

When HM runs on the following Python code:

```
def foo(a):
    if a:
        n = 1
        range(n)
        return n
    else:
        return None
```

It runs into some troubles. The `return` from the `True` branch sets the return type of `foo` to `int` but the one from the `False` branch sets it to `None`. How could we make this unification valid? Option types are generally described as a parametric type, `Optional[T0]`. To be able to unify `int` and `None`, we would instead need to unify `Optional[int]` and `None`, thus marking `n` as `Optional[int]`, which does not work, because `range` expects an `int`.

The solution we have adopted is to make type inference control-flow sensitive. When meeting an `if`, we generate a new copy of the variable environment for each branch, and we *merge* (not *unify*) the environments.

Likewise, if the condition is *explicitely* a check for `None`, as in:

```
if a is None:
    stuff()
else:
    return stuff(a)
```

the environment in the `True` branch holds the `None` type for `a`, and the `int` type in the `False` branch. This could be improved, as we support only a few patterns as the condition expression, there is something more generic to be done there.

This even led to improvement in our test base, as the following code was no longer correct:

```
def foo(x):
    v = x.get(1)
    return v + 1
```

Type inference computes that v is of type `Optional[T0]`, which is not compatible with `v + 1` and a `PythranTypeError` is raised. A compatible way to write this would be:

```
def foo(x):
    v = x.get(1)
    if v is None:
        pass  # or do stuff
    else:
        return v + 1
```

Handling Type Promotion

It's not uncommon to find this kind of code:

```
l = []
l.append(0)
l.append(3.14)
```

And there's nothing wrong with this in Python, but is this a type error for Pythran? In classical HM systems, that's a type error: `[]` is of type `List[T0]`, `list.append` is of type `Callable[[List[T0], T0], None]` so unification sets `T0` to `int` after first `append`, and fails upon the second `append` because unification between an `int` and a `float` fails.

Looking back in Python typing history, it seems that shedskin made the decision to consider it's not an error (see the blogpost announce on the topic. Several test cases of Pythran test suite would fail with a stricter typing, so let's try to achieve the same behavior as Shedskin, within HM.

The trick here is to consider a scalar as a tuple of four elements [0], one per scalar type we want to support. And then apply the following rule: the actual type of the scalar is the type of the first non variable type, starting from the lower index. Under that assumption,

- a `bool` is a `(T0, T1, T2, bool)`
- an `int` is a `(T0, T1, int, T2)`
- a `float` is a `(T0, float, T1, T2)`
- a `complex` is a `(complex, T0, T1, T2)`

When unifying an `int` with a `float`, regular unification yields `(T0, float, int, T2)` which is a `float` according to the previous definition.

If we want to enforce an `int`, say as argument of `range`, then we can define `strict_int` as `(no-complex, no-float, int, T0)` which still allows up-casting from `bool` to `int` but prevents up-casting from `int` to `float`.

Note

`numpy` introduces many sized type for integers, floating point numbers and complex numbers, with a set of rules to handle conversion between one and the other. As these conversions are generally possible in `numpy` (i.e. they dont raise a `TypeError`), we just use four scalar types: bool`, ``int, `complex` and `float`. `long` is merged into `int`, which also makes the Python2/3 compatibility easier.

Handling NDArray Type

`numpy.ndarray` is the corner stone of the `numpy` package. And it's super-flexible, allowing all kinds of broadcasting, reshaping, up-casting etc. Even if Pythran is far from supporting all of its features, it does support a wide set. The good news is that Pythran supports a lower version of `ndarray`, where the number of dimensions of an array does not change: it cannot be reshaped in place. For instance the C++ type returned by `numpy.ones((10, 10))` is `types::ndarray<double /*dtype*/, 2 /*nbdim*/>`.

We've extended the `typing` module to provide `NDArray`. For Pythran, the Python equivalent of the above C++ type is `NDArray[float, :, :]`.

And as we want it to be compatible with the way we defined an `Iterable`, an `NDArray` is actually a:

- `List[T0]` is considered as `(List, Sized, int, T0, T0)`
- `Dict[T0, T1]` is considered as `(Dict, Sized, T0, T1, T0)`
- ...
- `NDArray[complex, :]` is considered as `(Array, Sized, T0, complex, complex)`
- `NDArray[complex, :, :]` is considered as `(Array, Sized, T0, complex, NDArray[complex, :])`
- `NDArray[complex, :, :, :]` is considered as `(Array, Sized, T0, complex, NDArray[complex, :, :])`

That's a recursive definition, and that's pretty useful when used with our `MultiType` resolution. If we need to merge an `NDArray[complex, :, :]` and an `NDArray[complex, :, :, :]`, we end up with `(Array, Sized, T0, complex, (Array, Sized, T1, complex, T1))` which actually means *an array of complex with at least two dimensions*.

Testing the Brew

Let's be honest: the `tog` type system is more the result of tinkering than great research. Type systems is a complex field and I did my best to apply what I learned during my bibliography on the subject, but it still falls short in various places. So instead of a formal proof, here is some testing results :-).

First, the whole test suite passes without much modifications. It helped to spot a few *errors* in the tests, mostly code that was incorrect with respect to option types. We also updated the way we specify tests input type to rely on PEP484. A typical Pythran unit-test now looks like:

```
def test_shadow_import2(self):
    self.run_test(
```

```
        '''def shadow_import2(s):
               for set in s : set.add(1)''',
        [{1},{2}],
        shadow_import2=[List[Set[int]]
    )
```

where the `List[Set[int]]` expression describes the type for which the code must be instantiated.

The following code sample is adapted from the MyPy example page. It requires a type comment to be correctly typed, while Pythran correctly type checks it without annotation.

```
def wc(content):
    d = {}

    for word in content.split():
        d[word] = d.get(word, 0) + 1

    # Use list comprehension
    l = [(freq, word) for word, freq in d.items()]

    return sorted(l)
```

If we turn the `1` into `"1"`, we get the following error:

```
> pythran wc.py
CRITICAL You shall not pass!
E: Invalid operand for `+`: `int` and `str` (wc.py, line 5)
```

And if we remove the `0`, `d.get(word)` may return `None` and the error message becomes:

```
> pythran wc.py
CRITICAL You shall not pass!
E: Invalid operand for `+`: `Option[T0]` and `int` (wc.py, line 5)
```

Great!

Considering Numpy functions, we don't model all of them in tog, but we can still detect several interesting errors. For instance on a gaussian kernel (error-safe version from stackexchange):

```
import numpy as np
def vectorized_RBF_kernel(X, sigma):
    X2 = np.sum(np.multiply(X, X), 1) # sum colums of the matrix
    K0 = X2 + X2.T - 2 * X * X.T
    K = np.power(np.exp(-1.0 / sigma**2), K0)
    return K
```

```
def badcall(s):
    return vectorized_RBF_kernel(2, s)
```

Pythran correctly catches the error on `vectorized_RBF_kernel` call:

```
> pythran gaussian.py
CRITICAL You shall not pass!
E: Invalid argument type for function call to `Callable[[int, T3], ...]
`, tried Callable[[Array[1 d+, T0], T1], Array[1 d+, T2]] (gaussian.py,
 line 9)
```

Conclusion

I'm still not satisfied with the tog engine: it's relatively slow, not as accurate as I'd like it to be, and it's just a type checker: another (simpler) type engine is used to generate the actual C++ code. That's a lot of not very enthusiastic concluding remarks, but... I'm French :-)

On the good side, I happened to learn a *lot* about typing and about Python, while developing this. And Pythran is in a much better shape now, much more usable, easier to maintain too, so that was worth the effort :-)

Acknowledgments

As usual, I'd like to thanks Pierrick Brunet for all his help. He keeps feeding me with relevant insights, criticisms and great ideas. Thanks to OpenDreamKit for sponsoring that work, and in particular to Logilab for their support. Thanks to Lancelot Six, w1gz and Nicolas M. Thiéry for proof reading this post too :-)

And last, I'm in debt to all Pythran users for keeping the motivation high!

[0] That could be more actually, for instance to distinguish single precision float from double prcesion float, the `float32` and `float64` from numpy. But four types is enough for the envisonned type checking.

[1] The OOP style in C++ is not enforced by the Standard Library as much as it is in the Java SDK though.

Disclaimer: this report, together with its annexes and the reports for the earlier deliverables, is self contained for auditing and reviewing purposes. Hyperlinks to external resources are meant as a convenience for casual readers wishing to follow our progress; such links have been checked for correctness at the time of submission of the deliverable, but there is no guarantee implied that they will remain valid.