

Contents

Developer Manual	4
Milestones	4
Version Numbers	4
Current State	5
Coding Rules	5
Line length	5
Indentation	5
Identifiers	5
Module/Package Names	6
Prefer list contractions over "map", "filter", and "apply"	7
The "git flow" model	7
Design Descriptions	8
nuitka.Importing Module	8
Hooking for module import process	8
Language Conversions to make things simpler	9
The "assert" statement	9
The "comparison chain" expressions	9
The "execfile" builtin	9
Generator expressions with yields	9
Decorators	9
Inplace Assignments	10
Plan to replace "python-qt" for the GUI	10
Plan to add "ctypes" support	10
Goals/Allowances to the task	10
Type inference - The general Problem	10
Applying this to "ctypes"	13
Excursion to Functions	13
Excursion to Loops	14
Excursion to Conditions	15
Excursion to return statements	16
Excursion to yield statements	16
Mixed Types	16
Back to "ctypes"	16
Now to the interface	17
Discussing with examples	18
Code Generation Impact	19

Initial Implementation	19
Limitations for now	21
Realization	22
Idea Bin	23
Updates for this Manual	25

Developer Manual

The purpose of this developer manual is to present the current design of Nuitka, the coding rules, and the intentions of choices made. It is intended to be a guide to the source code, and to give explanations that don't fit into the source code in comments form.

It should be used as a reference for the process of planning and documenting decisions we made. Therefore we are e.g. presenting here the type inference plans before implementing them. And we update them as we proceed.x

It grows out of discussions and presentations made at PyCON alike conferences as well as private conversations or discussions on the mailing list.

Milestones

1. Feature parity with Python, understand all the language construct and behave absolutely compatible.

Feature parity has been reached for Python 2.6 and 2.7, we do not target any older CPython release. For Python 3.2, things are still not complete. The Python 3.x is not currently a high priority, but eventually we will get Nuitka going there too, and some of the basic tests already pass. You are more than welcome to volunteer for this task.

This milestone is considered reached.

2. Create the most efficient native code from this. This means to be fast with the basic Python object handling.

This milestone is considered mostly reached.

3. Then do constant propagation, determine as many values and useful constraints as possible at compile time and create more efficient code.

This milestone is considered in progress.

4. Type inference, detect and special case the handling of strings, integers, lists in the program.

This milestone is started only.

5. Add interfacing to C code, so Nuitka can turn a "ctypes" binding into an efficient binding as written with C.

This milestone is planned only.

6. Add hints module with a useful Python implementation that the compiler can use to learn about types from the programmer.

This milestone is planned only.

Version Numbers

For Nuitka we use defensive version numbering to indicate that it is not yet ready and useful for everything yet. We have defined milestones and the version numbers should express which of these, we consider done.

- So far:

Before milestone 1, we uses "0.1.x" version numbers. After reaching it, we used "0.2.x" version numbers.

- Now:

We currently use "0.3.x" version numbers as we still strive for milestone 2 and 3 to be really completed.

- Future:

When we start to have sufficient amount of type inference in a stable release, that will be "0.4.x" version numbers. With "ctypes" bindings in a sufficient state it will be "0.5.x".

- Final:

We will then round it up and call it "Nuitka 1.0" when this works as expected for a bunch of people. The plan is to reach this goal during 2012. This is based on lots of assumptions that may not hold up though.

Of course, this may be subject to change.

Current State

Nuitka top level works like this:

- "TreeBuilding" outputs node tree
- "Optimization" enhances it as best as it can
- "Finalization" marks the tree for code generation
- "CodeGeneration" creates identifier objects and code snippets
- "Generator" knows how identifiers and code is constructed
- "MainControl" keeps it all together

This design is intended to last. Regarding Types, the state is:

- Types are always "PyObject **", implicitly
- The only more specific use of type is "constant", which can be used to predict some operations, conditions, etc.
- Every operation is expected to have "PyObject **" as result, if it is not a constant, then we know nothing about it.

Coding Rules

These rules should generally be adhered when working on Nuitka code. It's not library code and it's optimized for readability, and avoids all performance optimizations for itself.

Line length

No more than 120 characters. Screens are wider these days, but most of the rules aim at keeping the lines below 90.

Indentation

No tabs, 4 spaces, no trailing white space.

Identifiers

Classes are camel case with leading upper case. Methods are with leading verb in lower case, but also camel case. Around braces, and after comma, there is spaces for better readability. Variables and parameters are lower case with "_" as a separator.

```
class SomeClass:

    def doSomething( some_parameter ):
        some_var = ( "lala", "lele" )
```

Base classes that are abstract end in "Base", so that a meta class can use that convention.

Function calls use keyword argument preferably. These are slower in CPython, but more readable:

```
return Generator.getSequenceCreationCode(
    sequence_kind      = sequence_kind,
    element_identifiers = identifiers,
    context             = context
)
```

The "=" are all aligned to the longest parameter names without extra spaces for it.

When the names don't add much value, sequential calls should be done, but ideally with one value per line:

```
return Identifier(
    "TO_BOOL( %s )" % identifier.getCodeTemporaryRef(),
    0
)
```

Here, "Identifier" will be so well known that the reader is expected to know the argument names and their meaning, but it would be still better to add them.

Contractions may span across multiple lines for increased readability:

```
result = [
    "PyObject *decorator_%d" % ( d + 1 )
    for d in
    range( decorator_count )
]
```

Module/Package Names

Normal modules are named in camel case with leading upper case, because their of role as singleton classes. The difference between a module and a class is small enough and in the source code they are also used similarly.

For the packages, no real code is allowed in them and they must be lower case, like e.g. "nuitka" or "codegen". This is to distinguish them from the modules.

Packages shall only be used to group packages. In "nuitka.codegen" the code generation packages are located, while the main interface is "nuitka.codegen.CodeGeneration" and may then use most of the entries as local imports.

The use of a global package "nuitka", originally introduced by Nicolas, makes the packaging of Nuitka with "distutils" etc. easier and lowers the requirements on changes to the "sys.path" if necessary.

Note

There are not yet enough packages inside Nuitka, feel free to propose changes as you see fit.

Names of modules should be plurals if they contain classes. Example is "Nodes" contains "Node" classes.

Prefer list contractions over "map", "filter", and "apply"

Using "map" and friends is considered worth a warning by "PyLint" e.g. "Used builtin function 'map'". We should use list comprehensions instead, because they are more readable.

List contractions are a generalization for all of them. We love readable and with Nuitka as a compiler will there won't be any performance difference at all.

I can imagine that there are cases where list comprehensions are faster because you can avoid to make a function call. And there may be cases, where map is faster, if a function must be called. These calls can be very expensive, and if you introduce a function, just for "map", then it might be slower.

But of course, Nuitka is the project to free us from what is faster and to allow us to use what is more readable, so whatever is faster, we don't care. We make all options equally fast and let people choose.

For Nuitka the choice is list contractions as these are more easily changed and readable.

Look at this code examples from Python:

```
class A:
    def getX( self ):
        return 1
    x = property( getX )

class B( A ):
    def getX( self ):
        return 2

A().x == 1 # True
B().x == 1 # True (!)
```

This pretty much is what makes properties bad. One would hope B().x to be "2", but instead it's not changed. Because of the way properties take the functions and not members, because they are not part of the class, they cannot be overloaded without re-declaring them.

Overloading is then not at all obvious anymore. Now imagine having a setter and only overloading the getter. How to you easily update the property?

So, that's not likable about them. And then we are also for clarity in these internal APIs too. Properties try and hide the fact that code needs to run and may do things. So let's not use them.

For an external API you may exactly want to hide things, but internally that has no use, and in Nuitka, every API is internal API. One exception may be the "hints" module, which will gladly use such tricks for easier write syntax.

The "git flow" model

- The flow was used for the a couple of releases and subsequent hotfixes.

A few feature branches were used so far. It allows for quick delivery of fixes to both the stable and the development version, supported by a git plugin, that can be installed via "apt-get install git-flow" on latest Debian Testing at least.

- Stable (master branch)

The stable version, is expected to pass all the tests at all times and is fully supported. As soon as bugs are discovered, they are fixed as hotfixes, and then merged to develop by the "git flow" automatically.

- Development (develop branch)

The future release, supposedly in almost ready for release state at nearly all times, but this is as strict. It is not officially supported, and may have problems and at times inconsistencies.

- **Feature Branches**

On these long lived developments that extend for multiple release cycles or contain changes that break Nuitka temporarily. They need not be functional at all.

Current Feature branches:

- "feature/minimize_CPython26_tests_diff": Maximizing compatibility, we minimize the differences to baseline CPython2.6 tests. Currently stuck at "test_inspect.py" and recently fallen behind, to be continued once Kay is free from preparatory works for "feature/ctypes_annotation" branch work.
- "feature/ctypes_annotation": Achieve the inlining of ctypes calls, so they become executed at no speed penalty compared to direct calls via extension modules. This being fully CPython compatible and pure Python, is considered the "Nuitka" way of creating extension modules that provide bindings.

Design Descriptions

These should be a lot more and contain graphics from presentations given. It will be filled in, but not now.

nuitka.Importing Module

- From the module documentation

The actual import of a module may already execute code that changes things. Imagine a module that does "os.system()", it will be done. People often connect to databases, and these kind of things, at import time. Not a good style, but it's being done.

Therefore CPython exhibits the interfaces in an "imp" module in standard library, which one can use those to know ahead of time, what file import would load. For us unfortunately there is nothing in CPython that is easily accessible and gives us this functionality for packages and search paths exactly like CPython does, so we implement here a multi step search process that is compatible.

This approach is much safer of course and there is no loss. To determine if it's from the standard library, one can abuse the attribute "__file__" of the "os" module like it's done in "isStandardLibraryPath" of this module.

- Role

This module serves the recursion into modules and analysis if a module is a known one. It will give warnings for modules attempted to be located, but not found. These warnings are controlled by a while list inside the module.

Hooking for module import process

Currently, in created code, for every "import" variable a normal "__import__()" call is executed. The "ExeModuleUnfreezer.cpp" (located in "nuitka/build/static_src") provides the implementation of a "sys.meta_path" hook.

This one allows us to have the Nuitka provided module imported even when imported by non-compiled code. Kay learned this at PyCON DE conference, from a presentation by the implementer of that PEP, and it's very useful, as it increased compatibility over the previous approach of special casing imports to check if it's the included module.

Note

Of course it would make sense to compile time detect which module it is that is being imported and then to make it directly. At this time, we don't have this inter-module optimization yet, it should be easy to add.

Language Conversions to make things simpler

There are some cases, where the Python language has things that can in fact be expressed in a simpler or more general way, and where we choose to do that at either tree building or optimization time.

The "assert" statement

Handling is:

This makes assertions the same as a branch guarded exception, what it really is, and removes the need for any special code or optimizations to concern with it.

This transformation is performed at tree building already.

The "comparison chain" expressions

This transformation is performed at tree building already. The assignment expressions are not standard Python, but a useful addition that enables this transformation and to express the short circuit nature of comparison chains.

The "execfile" builtin

Handling is:

Note

This allows optimizations to discover the file opening nature easily and apply file embedding or whatever we will have there one day.

This transformation is performed when the "execfile" builtin is detected as such during optimization.

Generator expressions with yields

These are converted at tree building time into a generator function body that yields the iterator given, which is the put into a for loop to iterate, created a lambda function of and then called with the first iterator.

That eliminates the generator expression for this case. It's a bizarre construct and with this trick needs no special code generation.

Decorators

When one learns about decorators, you see that:

The only difference is the assignment to function. In the "@decorator" case, if the decorator fails with an exception, the name "function" is not assigned. Internally in Nuitka this assignment is therefore from a "function body expression" and only the last decorator returned value is assigned to the function name.

This removes the need for code generation to support decorators. And it should make the two variants optimize equally well.

Inplace Assignments

Inplace assignments are be re-formulated to an expression using temporary variables.

These are not as much a reformulation of "+=" to "+", but instead one which makes it explicit that the assign target may change its value.

```
a += b
```

```
_tmp = a.__iadd__( b )  
  
if a is not _tmp:  
    a = _tmp
```

Using "__iadd__" here to express that not the "+", but the in-place variant "iadd" is used instead. The "is" check may be optimized away depending on type and value knowledge later on.

Plan to replace "python-qt" for the GUI

Porting the tree inspector available with "--dump-gui" to "wxwindows" is very much welcome as the "python-qt4" bindings are severely under documented.

Plan to add "ctypes" support

Add interfacing to C code, so Nuitka can turn a "ctypes" binding into an efficient binding as if it were written manually with Python C-API or better.

Goals/Allowances to the task

1. Goal: Must not use any existing C/C++ language file headers, only generate declarations in generated C++ code ourselves. We would rather write a C header to "ctypes" declarations convert if it needs to be.
2. Allowance: May use "ctypes" module at compile time to ask things about "ctypes" and its types.
3. Goal: We should make use that allowance to use "ctypes", to e.g. not hard code what "ctypes.c_int()" gives, unless there is a specific benefit.
4. Allowance: Not all "ctypes" usages must be supported immediately.
5. Goal: Try and be as general as possible. For the compiler, "ctypes" support should be hidden behind a generic interface of some sort. Supporting "math" module should be the same thing.

Type inference - The general Problem

Part of the goal is to forward value knowledge. When you have "a = b", that means that a and b now "alias". And if you know the value of "b" you can assume to know the value of "a".

If that value is a constant, you will want to push it forward, because storing the constant has a cost and loading the variable as well. So, you want to be able collapse such code:

```
a = 3  
b = 7  
c = a / b
```

to:

```
c = 3 / 7
```

and that obviously to:

```
c = 0
```

This may be called (Constant) Value Propagation. But we are aiming for more. In order to fully benefit from type knowledge, the new type system must be able to be fully friends with existing builtin types. The behavior of a type "long", "str", etc. ought to be implemented as far as possible with the builtin "long", "str" as well.

Note

This "use the real thing" concept extends beyond builtin types, "ctypes.c_int()" should also be used, but we must be aware of platform dependencies. The maximum size of "ctypes.c_int" values would be an example of that. Of course that may not be possible for everything.

This approach has well proven itself with builtin functions already, where we use real builtins where possible to make computations. We have the problem though that builtins may have problems to execute everything with reasonable compile time cost.

```
len( "a" * 10000000000000 )
```

To predict this code, calculating it at compile time using constant operations, while feasible, puts an unacceptable burden on the compilation.

Esp. we wouldn't want to produce such a constant and stream it, the C++ code would be too huge. So, we need to stop the "*" operator from being used at compile time and live with reduced knowledge, already here:

```
"a" * 100000000000000
```

Instead, we would probably say that for this expression:

- The result is a "str" or "PyStringObject".
- We know its length exactly, it's "100000000000000".
- Can predict every of its elements when index, sliced, etc., if need be, with a function.

Similar is true for this nice thing:

```
range( 100000000000000 )
```

So it's a rather general problem, this time we know:

- The result is a "list" or "PyListObject"
- We know its length exactly, "100000000000000"
- Can predict every of its elements when index, sliced, etc., if need be, with a function.

Again, we wouldn't want to create the list. Nuitka currently refuses to compile time calculate lists with more than 256 elements, which is an arbitrary choice.

Note

We could know, from use of the "range" result maybe, that we ought to prefer a "xrange", but that's not as much useful except maybe at code generation time. But we would rather benefit from knowing we need not have any such object at all to satisfy e.g. loop conditions.

Note

In our builtin code, we have specialized "range()" to check for the result size in a prediction. This ought to be generalized and take the computation cost and result size into account.

Now lets look at a use:

```
for x in range( 10000000000000 ):
    doSomething()
```

Looking at this example, one way to look at it, would be to turn "range" into "xrange", note that "x" is unused. But in fact, what would be best, is to notice that "range()" generated value is not really used, but only the length of the expression matters. And even if "x" were used, only the ability to predict the value from a function would be interesting, so we would use that computation function instead.

Predict from a function could mean to have Python code to do it, as well as C++ code to do it. Then code for the loop can be generated without any CPython usage at all.

Note

Of course, it would only make sense where such calculations are "O(1)" complexity, i.e. do not require recursion like "n!" does.

The other thing is that CPython appears to take length hints from objects for some operations, and there it would help too, to track length of objects, and provide it, to outside code.

Back to the original example:

```
len( "a" * 10000000000000 )
```

The theme here, is that when we can't pre-compute all intermediate expressions, and we sure can't always in the general case. But we can still, predict some of properties of an expression result, more or less.

Here we have "len" to look at an argument that we know the size of. Great. We need to ask if there are any side effects, and if there are, we need to maintain them of course, but generally this appears feasible, and is already being done by existing optimizations if an operation generates an exception.

Applying this to "ctypes"

The not so specific problem to be solved to understand "ctypes" declarations is maybe as follows:

```
import ctypes
```

This leads to Nuitka tree containing an "import module expression", that can be predicted by default to be a module object, and it can be better known as "ctypes" from standard library with more or less certainty. See the section about "Importing".

So that part is easy, and it's what should happen. During optimization, when the module import expression is examined, it should say:

- "ctypes" is a module
- "ctypes" is from standard library (if it is, may not be true)
- "ctypes" has a "ModuleFriend" that knows things about it attributes, that should be asked.

The later is the generic interface, and the optimization should connect the two, of course via package and module full names. It will need a "ModuleFriendRegistry", from which it can be pulled. It would be nice if we can avoid "ctypes" to be loaded into Nuitka unless necessary, so these need to be more like a plug-in, loaded only if necessary.

Coming back to the original expression, it also contains an assignment expression, because it is more like this:

```
ctypes = __import__( "ctypes" )
```

The assigned to object, simply gets the type inferred propagated, and the question is now, if the propagation should be done as soon as possible and to what, or later.

For variables, we don't currently track at all any more than there usages read/write and that is it. And we are not good at it either.

The problem with tracking it, is that such information may continuously become invalid at many instances, and it can be hard to notice mistakes due to it. But if do not have it fixed, how to we detect this:

```
ctypes.c_int()
```

How do we tell that "ctypes" is at that point a variable of module object or even the ctypes module, and that we know what it's "c_int" attribute is, and what it's call result is.

We should therefore, forward the usage of all we know and see if we hit any "ctypes.c_int" alike. This is more like a value forward propagation than anything else. In fact, constant propagation should only be the special case of it.

Excursion to Functions

In order to decide what is best, forward or backward, we consider functions. If we propagate forward, how to handle this:

```
def my_append( a, b ):
    a.append( b )

    return a
```

We would notate that "a" is first a "PyObject parameter object", then something that has an "append" attribute, when returned. The type of "a" changes after "a.append" lookup succeeds. It might be an object, but e.g. it could have a higher probability of being a "PyListObject".

Note

If classes in the program have an "append" attribute, it should play a role too, there needs to be a way to plug-in to this decisions.

This is a more global property of "a" value, and true even before the append succeeds, but not as much maybe, so it would make sense to apply that information after an analysis of all the node. This may be "Finalization" work.

```
b = my_append( [], 3 )  
  
assert b == [3] # Can be known now
```

Goal: The structure we use should make it easy to visit "my_append" and then have something that easily allows to plug in the given values and know things. We need to be able to tell, if evaluating "my_append" makes sense with given parameters or not.

We should e.g. be able to make "my_append" tell, one or more of these:

- Returns the first parameter value (unless it raises an exception)
- The return value has the same type as "a" (unless it raises an exception)

It would be nice, if "my_append" had information, we could instantiate with "list" and "int" from the parameters, and then e.g. know what it does in that case.

Doing it "forward" appears to be best suited for functions and therefore long term. We will try it that way.

Excursion to Loops

```
a = 1  
  
for i in range( 10 ):  
    b = a + 1  
    a = b  
  
print a
```

The handling of "for" (and "while") loops has its own problem. They are similar to that of function calls and their bodies. The "for" loop would have a start assumption that "a" is constant, but that is only true for the first iteration. So, we can't pass knowledge from outside the for loop directly into the for loop body.

We will do a first pass, where we collect invalidations of the outside knowledge. The assignment to "a" should make it an alternative with what we knew about "b". And we can't really assume to know anything about a to e.g. predict "b" due to that. That first pass needs to scan for assignments, and treat them as invalidations.

Excursion to Conditions

```
if cond:
    x = 1
else:
    x = 2

b = x < 3
```

The above code contains a condition, and these have the problem, that when exiting the conditional block, it must be clear to the outside, that things changed inside the block may not necessarily apply. Even worse, one of 2 things might be true. In one branch, the variable "x" is constant, in the other too, but it's a different value.

So for constants, we need to have the constraint collection know when it enters a conditional branch, and when it does, it must take special precautions, to preserve the existing state. When exiting all the branches, these branches must be merged, with new information.

In the above case:

- The "yes" branch knows variable "x" is an "int" of constant value "1"
- The "no" branch knows variable "x" is an "int" of constant value "2"

That should be collapsed to:

- The variable "x" is an integer of value in "(1,2)"

When should allow to precompute the value of this:

```
b = x < 3
```

The comparison operator can work on the function that provides all values in see if the result is always the same. Because if it is, and it is, then it can tell:

- The variable "b" is a boolean of constant value "True".

For conditional statements optimization, the following is note-worthy:

- The value of the condition is known to pass truth check or not inside either branch.

We may want to take advantage of it. Consider e.g.

```
if type( a ) is list:
    a = a.append( x )
else:
    a += ( x, )
```

In this case, the knowledge that "a" is a list, could be used to generate better code and with definite knowledge that "a" is of type list. These is a lot more to do, until we understand "type checks" though.

- If 2 branches exist, or one makes a difference.

If both branches exist, both should fork existing state and continue it, and afterwards merge those 2 and replace the state before the statement.

If only one branch exist, that one should fork existing state and continue it, but afterwards, it needs to be merged back to the state before the statement.

Excursion to return statements

The return statement ought to be the last statement of inspected block, or else previous optimization steps have failed. With a conditional statement branch, in case it ends with a return statement, the merging of the constraint collection must consider it by not taking any knowledge from it at all.

If all branches of a conditional statement return, that should have an optimization to discover that, and remove the following statements. Currently no such optimization exists and it should be added, so value propagation can rely on it to be handled already.

Excursion to yield statements

The yield statement can be treated like a normal function call, and as such invalidates some known constraints.

Mixed Types

Consider the following inside a function or module:

```
if cond is not None:
    a = [ x for x in something() if cond(x) ]
else:
    a = ()
```

A programmer will often not make a difference between "list" and "tuple". In fact, using a tuple is a good way to express that something won't be changed later, as these are mutable.

Note

Better programming style, would be to use this:

```
if cond is not None:
    a = tuple( x for x in something() if cond(x) )
else:
    a = ()
```

People don't do it, because they dislike the performance hit encountered by the generator expression being used to initialize the tuple. But it would be more consistent, and so Nuitka is using it, and of course one day ought to be able to evaluate the tuple call to the generator expression by the means of a "tuple contraction".

To Nuitka though this means, that if "cond" is not predictable, after the conditional statement we may either have a "tuple" or a "list". In order to represent that without resorting to "I know nothing about it", we need a kind of "min/max" operating mechanism that is capable of say what is common with multiple alternative values.

Back to "ctypes"

```
v = ctypes.c_int()
```

Coming back to this example, we needed to propagate "ctypes", then we can propagate "something" from "ctypes.int" and then known what this gives with a call and no arguments, so the walk of the nodes, and diverse operations should be addressed by a module friend.

In case a module friend doesn't know what to do, it needs to say so by default. This should be enforced by a base class and give a warning or note.

Now to the interface

The following is the intended interface

- Base class "ValueFriendBase" according to rules.

The base class offers methods that allow to check if certain operations are supported or not. These can always return "True" (yes), "False" (no), and "None" (cannot decide). In the case of the later, optimizations may not be able do much about it. Lets call these values "tristate".

Part of the interface is a method "getComputation" which gives the node the chance to return a tristate together with a lambda, that when executed, produces either an exception or constant value.

The "getComputation" may be able to produce exceptions or constant even for non-constant inputs depending on the operation being performed. For every computational step it will be executed.

In this sense, attribute lookup is also a computation, as its value might be computed as well. Most often an attribute lookup will produce a new value, which is not assigned, but e.g. called. In this case, the call value friend may be able to query its called expression for the attribute call prediction.

- Name for module "ValueFriends" according to rules.

These should live in a package of some sort and be split up into groups later on, but for the start it's probably easier to keep them all in one file.

- Class for module import expression "ValueFriendImportModule".

This one just knows that something is imported and not how or what it is assigned to, it will be able in a recursive compile, to provide the module as an assignment source, or the module variables or submodules as an attribute source.

- Class for module value friend "ValueFriendModule".

The concrete module, e.g. "ctypes" or "math" from standard library.

- Base class for module and module friend "ValueFriendModuleBase".

This is intended to provide something to overload, which e.g. can handle "math" in a better way.

- Module "ModuleFriendRegistry"

Provides a register function with "name" and instances of "ValueFriendModuleBase" to be registered. Recursed to modules should integrate with that too. The registry could well be done with a metaclass approach.

- The module friends should each live in a module of their own.

With a naming policy to be determined. These modules should add themselves via above mechanism to "ModuleFriendRegistry" and all shall be imported and register. Importing of e.g. "ctypes" should be delayed to when the friend is actually used. A meta class should aid this task.

The delay will avoid unnecessary blot of the compiler at run time, if no such module is used. For "qt" and other complex stuff, this will be a must.

- A collection of "ValueFriend" instances expresses the current data flow state.

- This collection should carry the name "ConstraintCollection"

- Updates to the collection should be done via methods
 - onAssignmentToTargetFromValueFriend(target, value_friend)"
 - "onAttributeLookup(source, attribute_name)"
 - "onOutsideCode()"
 - "passedByReference(var_name)"
 - etc. (will decide the actual interface of this when implementing its use)

- This collection is the input to walking the tree by "execute", i.e. per module body, per function body, per loop body, etc.
- The walk should initially be single pass, that means it does not maintain the history.

Note

Warning

With this, the order of node walking becomes vital to correctness. The evaluation order of the generated code is now absolutely needed.

This may carry bug potential. We will need tests that cover this.

Discussing with examples

The following examples:

```
# Assignment, the source decides the type of the assigned expression
a = b

# Operator "attribute lookup", the looked up expression decides via its "ValueFriend"
ctypes.c_int

# Call operator, the called expressions decides with help of arguments, which may
# receive value friends after walking to them too.
called_expression_of_any_complexity()

# import gives a module any case, and the "ModuleRegistry" may say more.
import ctypes

# From import need not give module, "x" decides
from x import y

# Operations are decided by arguments, and CPython operator rules between argument
# "ValueFriend"s.
a + b
```

The walking of the tree is best done in "Optimization" and can be used to implement many optimizations in a more consistent and faster way. We currently check the tree for calls to builtins with constant arguments. But with the new way of walking, we reverse the order of the check.

Now:

- Check all tree for suitable "builtin" function calls
- Check their arguments if they are constants
- Replace builtin call node with constant results or known exception

Future:

- Walk the tree and enter arguments of builtin function calls
- Collect knowledge about the argument, including maybe that they are constant or known length
- Ask the "BuiltinValueFriend" what it can say. If it says the value is of constant value, replace the node with constant results or known exception

It's really exciting to see, how this proposal cleans up the existing code and integrates with it.

Code Generation Impact

Right now, code generation assumes that everything is a Python object, and does not take "int" or these at all, and it should remain like that for some time to come.

Instead, "ctypes" value friend will be asked give "Identifiers", like other codes do too from calls. And these need to be able to convert themselves to objects to work with the other things.

But Code Generation should no longer require that operations must be performed on that level. Imagine e.g. the following calls:

```
c_call( other_c_call() )
```

Value return by other_c_call() of say "c_int" type, should be possible to be fed directly into another call. That should be easy by having a "asIntC()" in the identifier classes, which the "ctypes" Identifiers handle without conversions.

Code Generation should one day also become able to tell that all uses of a variable have only "c_int" value, and use "int" instead of "PyObjectLocalVariable" directly, or at least a "PyIntLocalVariable" of similar complexity as "int" after the C++ compiler performed its inlining.

Such decisions would be prepared by finalization, which then would track the history of values throughout a function or part of it.

Initial Implementation

The "ValueFriendBase" interface will be added to `_all_` expressions nodes creation time, a node may either do it for itself (constant reference is an obvious example) or may delegate the task to an instantiated object of "ValueFriendBase" inheritance.

Initially most of them will only be able to give up on about anything. And it will be little more than a tool to do lookups.

It will then be the first goal to turn the following code into better performing one:

```
a = 3
b = 7
c = a / b
```

to:

```
a = 3
b = 7
c = 3 / 7
```

The assignments to "a" and "b" might become prey to "unused" assignment analysis later on, but that is not important yet. Also "3 / 7" could be optimized while going through it, but there is already code that does this "OptimizeConstantOperations" easily. So that would be a later step.

Note

This part is implemented.

Then second goal is to understand all of this:

```
def f():
    a = []

    print a

    for i in range(1000):
        print a

        a.append( i )

    return len( a )
```

Note

There are many operations in this, and all of them should be properly handled, or at least ignored in safe way.

The first goal code gave us that the "list" has an annotation from the assignment of "[]" and that it will be copied to "a" until the for loop is encountered. Then it must be removed, because the "for" loop somehow says so.

The "a" may change its value, due to the unknown attribute lookup of it already, not even the call. The for loop must be able to say "may change value" due to that, of course also due to the call of that attribute too.

The code should therefore become equivalent to:

```
def f():
    a = []

    print []

    for i in range(1000):
        print a

        a.append( i )

    return len( a )
```

But no other changes must occur, especially not to the "return" statement, it must not assume "a" to be constant "[]" but an unknown "a" instead.

With that, we would handle this code correctly and have some form constant value propagation in place, handle loops at least correctly, and while it is not much, it is an important demonstration of the concept.

Note

This part is implemented.

The third goal is to understand the following:

```

def f( cond ):
    y = 3

    if cond:
        x = 1
    else:
        x = 2

    return x < y

```

In this we have a branch, and we will be required to keep track of both the branches separately, and then to merge with the original knowledge. After the conditional statement we will know that "x" is an "int" with possible values in "(1,2)", which can be used to predict that the return value is always "True".

The forth goal will therefore be that the "ValueFriendConstantList" knows that append changes "a" value, but it remains a list, and that the size increases by one. It should provide an other value friend "ValueFriendList" for "a" due to that.

In order to do that, such code must be considered:

```

a = []

a.append( 1 )
a.append( 2 )

print len( a )

```

It will be good, if "len" still knows that "a" is a list, but not the constant list anymore.

From here, work should be done to demonstrate the correctness of it with the basic tests applied to discover undetected issues.

Fifth and optional goal: Extra bonus points for being able to track and predict "append" to update the constant list in a known way. Using "list.append" that should be done and lead to a constant result of "len" being used.

The sixth and challenging goal will be to make the code generation be impacted by the value friends types. It should have a knowledge that "PyList_Append" does the job of append and use "PyList_Size" for "len". The "ValueFriends" should aid the code generation too.

Last and right now optional goal will be to make "range" have a value friend, that can interact with iteration of the for loop, and "append" of the "list" value friend, so it knows it's possible to iterate 5000 times, and that "a" has then after the "loop" this size, so "len(a)" could be predicted. For during the loop, about a the range of its length should be known to be less than 5000. That would make the code of goal 2 completely analyzed at compile time.

Limitations for now

- The collection of value friends will not have a history and be mutated as the processing goes.

We will see, if we need any better at all. One day we might have passes with more expensive and history maintaining variants, that will be able to look at one variable and decide "value is only written, never read" and make something out of it.

- Only enough to trace "ctypes" information through the code

We won't cover everything immediately. We need to consider re-factoring existing optimizations into such that happen during the pass with value information. The builtins have already been mentioned as a worth-while target. It would also validate the new design. But it should not block to reach the ability to implement "ctypes".

- Aim only for limited examples. For "ctypes" that means to compile time evaluate:

```
print ctypes.c_int( 17 ) + ctypes.c_long( 19 )
```

Later then call to "libc" or something else universally available, e.g. "strlen()" or "strcmp()" from full blown declarations of the callable.

- We won't have the ability to test that optimizations are actually performed, we will check the generated code by hand.

With time, Kay will add XML based checks with "xpath" queries, expressed as hints, but that is some work that will be based on this work here. The "hints" fits into the "ValueFriends" concept nicely or so the hope is.

- No inter-function optimization functions yet

It's not needed yet or so we think. Of course, once in place, it will make the "ctypes" annotation even more usable. Using "ctypes" objects inside functions, while creating them on the module level, is therefore not immediately going to work.

- No loops yet

Loops break value propagation. For the "ctypes" use case, this won't be much of a difficulty. Due to the strangeness of the task, it should be tackled later on at a higher priority.

- Not too much.

Try and get simple things to work now. We shall see, what kinds of constraints really make the most sense. Understanding "list" subscript/slice values e.g. is not strictly useful for much code and should not block us.

Note

This new design is not the final one likely, it just needs to be better than existing optimizations design.

Realization

Kay will attempt to provide the framework parts that provide the interface and Christopher will work on the "ctypes" as an example.

The work is likely to happen on a git feature branch named "ctypes_annotation". It will likely be long lived, and Kay will move usable bits out of it for releases, and occasional "git flow feature rebase" at agreed times.

Note

After handing over the work in a usable state, Kay will focus on allowing other developers to push branches like these at their own discretion and with some form of git commit emails for better collaboration. In the mean time, "git format-patch" will do.

Idea Bin

This an area where to drop random ideas on our minds, to later sort it out, and out it into action, which could be code changes, plan changes, issues created, etc.

- Faster/cheaper class creation for the normal case.

Right now for every class body, there is a "MAKE_CLASS_*" with frame guard, exception keeper, etc. overhead, but for most classes that is not needed at all. Most often the building of functions is all that happens, if at all. For these cases, a different approach might be taken, that is to simply build the dictionary directly if possible and no side-effect could happen.

A finalization step ought to markup classes whose dictionary elements only have things without side effects, and building functions isn't that. Side effects may also be OK, if order is respected, we probably mean "conditions" here that are not conditional expressions or can be reduced to such.

- The conditional expression needs to be handled like conditional statement for propagation.

We branch conditional statements for value propagation, and we likely need to do the same for conditional expressions too. May apply to "or" as well, and "and", because there also only conditionally code is executed.

Is there any re-formulation of conditional expressions with "and" and "or" that is generally true?

- Generator expressions should be re-formulated as functions.

Generally they could be turned into nested creations of for loop function bodies with the first iterator as an argument. Right now, that would loose their optimizations, but once we could recognize cases, where that optimization could be applied from the code at code generation time, we wouldn't have to carry them through optimizations anymore, so that idea is worth perusing.

- For loops should probably become while loops.

Instead of treating for loops special, we could consider this:

```
for x,y in iterable:
    if something( x ):
        break
else:
    otherwise()
```

```
_broken = False
while True: # forever
    try:
        _tmp = next( iterable )
        x, y = _tmp
    except StopIteration:
        break

    if something ( x ):
        _broken = True
        break

if not _broken:
    otherwise()
```

It's a reduction of for loop to a while loop. But we would loose all chances to detect the loop exit conditions, but did we have it? This is also very similar to the C++ code we generate for them right now. The detection of "iterable", how often next will be possible at max, would e.g. be more difficult, but also more general.

Not visible here, is how "_tmp" is released right after unpacking, and how the "StopIteration" need not be a real exception, but is a "NULL" return of "ITERATOR_NEXT" for maximum efficiency. A structure will be needed that holds "tmp", much like a "with" block.

- With statements should become scoped

The algorithm of "with" statements should be re-formulated in the node tree. The taking and calling of "__enter__" and "__exit__" with arguments, should be presented there in order to be absolutely safe.

- Assignments unpacking should use scoped temporary variables

```
a, b.attr, c[ind] = d = e, f, g = h()
```

Can become this:

```
_tmp = h()

_iter1 = iter( tmp )
_tmp1, _tmp2, tmp3 = unpack_and_check( _iter1, 3 )
a = _tmp1
b.attr = _tmp2
c[ind] = _tmp3
d = _tmp

_iter2 = iter( tmp )
_tmp4, _tmp5, tmp6 = unpack_and_check( _iter2, 3 )
e = _tmp2
f = _tmp3
g = _tmp
```

That will of course be a lot less inefficient, until we can value propagate as well as in the past, which was working for constants. In these cases, the "iter" taking and the "unpack_and_check" calls can be optimized away again.

- Code Templates may become objects.

It should only wrap around the "%" operator and provide the ability to display the template name in tracebacks as well as in generated code. So one could optionally enable things and know from what template a code snippet comes.

Maybe they should overload "%" to start with it easily. And the code template could be the doc string of the class for simplicity.

Updates for this Manual

This document is written in REST. That is an ASCII format readable as ASCII, but used to generate a PDF or HTML document.

You will find the current source under:
http://nuitka.net/gitweb/?p=Nuitka.git;a=blob_plain;f=Developer_Manual.txt

And the current PDF under: http://nuitka.net/doc/Developer_Manual.pdf