Contents

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.

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 CPython, understand all the language construct and behave absolutely compatible.

   Feature parity has been reached for CPython 2.6 and 2.7. We do not target any older CPython release. For CPython 3.2 it also has been reached. We do not target older CPython 3.1 and 3.0 releases.

   This milestone was reached.

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

   This milestone was 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 almost reached.

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

   This milestone is considered in progress

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 used "0.1.x" version numbers. After reaching it, we used "0.2.x" version numbers.

  Before milestone 2 and 3, we used "0.3.x" version numbers. After almost reaching 3, and beginning with 4, we use "0.4.x" version numbers.

- Future:

When we start to have sufficient amount of type inference in a stable release, that will be "0.5.x" version numbers. With `ctypes` bindings in a sufficient state it will be "0.6.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 2014. 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 code aims at keeping the lines below 100.

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 = ( "foo", "bar" )
```

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 should 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 lets 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/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.

Checking the Source

The checking for errors is currently done with "PyLint". In the future, Nuitka will gain the ability to present its findings in a similar way, but this is not a priority, and not there yet.

So, we currently use "PyLint" with options defined in a script.

```
./misc/check-with-pylint --hide-todos
```

Ideally the above command gives no warnings. This has not yet been reached. The existing warnings serve as a kind of "TODO" items. We are not white listing them, because they indicate a problem that should be solved.

If you submit a patch, it would be good if you checked that it doesn't introduce new warnings, but that is not strictly required. it will happen before release, and that is considered enough. You probably are already aware of the beneficial effects.

Running the Tests

This section describes how to run Nuitka tests.

Running all Tests

The top level access to the tests is as simple as this:

```
./misc/check-release
```

For fine grained control, it has the following options:

```
-h, --help            show this help message and exit
--skip-basic-tests    The basic tests, execute these to check if Nuitka is
                      healthy. Default is True.
--skip-syntax-tests   The syntax tests, execute these to check if Nuitka
                      handles Syntax errors fine. Default is True.
--skip-program-tests  The programs tests, execute these to check if Nuitka
                      handles programs, e.g. import recursions, etc. fine.
                      Default is True.
--skip-reflection-test
```

```
                      The reflection test compiles Nuitka with Nuitka, and
                      then Nuitka with the compile Nuitka and compares the
                      outputs. Default is True.
--skip-cpython26      The standard CPython2.6 test suite. Execute this for
                      all corner cases to be covered. With Python 2.7 this
                      covers exception behavior quite well. Default is True.
--skip-cpython27      The standard CPython2.7 test suite. Execute this for
                      all corner cases to be covered. With Python 2.6 these
                      are not run. Default is True.
```

You will only run the CPython 2.6 test suite, if you have the submodules of the Nuitka git repository checked out. Otherwise, these will be skipped automatically with a warning that they are not available.

> ***Note***
>
> The CPython 2.7 test suite is not even public yet as it should also first undergo a "minimize diff" activity, before doing that. I didn't take the time for that yet, but I intend to do it. This is of course important for set and dict contractions.

The policy is generally, that "./misc/check-release" running and passing all tests shall be considered sufficient for a release.

Basic Tests

You can run the "basic" tests like this:

```
./tests/basics/run_all.py search
```

These tests normally give sufficient coverage to assume that a change is correct, if these tests pass. To control the Python version used for testing, you can set the `PYTHON` environment variable to e.g. "python3.2", or execute the "run_all.py" with the intended version, it is portable across all supported Python versions.

Syntax Tests

Then there are "syntax" tests, i.e. language constructs that need to give a syntax error.

It sometimes happens that Nuitka must do this itself, because the `ast.parse` don't see the problem. Using `global` on a function argument is an example of this. These tests make sure that the errors of Nuitka and CPython are totally the same for this:

```
./tests/syntax/run_all.py search
```

Program Tests

Then there are small programs tests, that exercise all kinds of import tricks and problems with inter-module behavior. These can be run like this:

```
./tests/programs/run_all.py search
```

Compile Nuitka with Nuitka

And there is the "compile itself" or "reflected" test. This test makes Nuitka compile itself and compare the resulting C++, which helps to find indeterminism. The test compiles every module of Nuitka into an extension module and all of Nuitka into a single binary.

That test case also gives good coverage of the `import` mechanisms, because Nuitka uses a lot of packages.

```
./tests/reflected/compile_itself.py
```

Design Descriptions

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

Choice of the Target Language

- Choosing the target language, is an important decision
  - The portability of Nuitka is decided here
- Other factors:
  - How difficult is it to generate the code?
  - Does the Python C-API have bindings?
  - Is that language known?
  - Does the language aid to find bugs?
- These candidates were considered
  - C++03, C++11, Ada

Requirement to Language matrix:

Requirement\Language	C++03	C++11	Ada
Portable	Yes	No [1]	Yes
Knowledge	Yes	No [2]	Yes
Python C-API	Yes	Yes	No [3]
Runtime checks	No	No	Yes [4]
Code Generation	Hard	Easy	Harder

1:: C++11 is not fully supported from any compiler (temporary problem)

2:: Not a whole lot of people have C++11 knowledge. My *only* C++11 Code was that in Nuitka.

3:: The Python C-API for Ada would have to be created by us, possible just big project by itself.

4:: Runtime checks exist only for Ada in that quality. I miss automatic `CONSTRAINT_ERROR` exceptions, for data structures with validity indicators, where in other languages, I need to check myself.

The *decision for C++03* is ultimately:

- for portability
- for language knowledge

All of these are important advantages.

For C++11 initially spoke easy code generation.

- variadic templates
- raw strings

Yet, as it turns out, variadic templates do not help with evaluation order, so that code that used it, needed to be changed to generating instances of their code. And raw strings turned out to be not as perfect as one wants to be, and solving the problem with C++03 is feasible too, even if not pretty.

For Ada would have spoken the time savings through run time checks, which would have shortened some debugging sessions quite some. But building the Python C-API bindings on our own, and potentially incorrectly, would have eaten that up.

Locating Modules and Packages

The search for of modules used is driven by `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 "ModuleUnfreezer.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 had 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.

Supporting `__class__` of Python3

In Python3 the handling of `__class__` and `super` is different from Python2. It used to be a normal variable, and now the following things have changed.

- The use of the `super` variable name triggers the addition of a closure variable `__class__`, as can be witnessed by the following code:

```
class X:
   def f1( self ):
      print( locals() )

   def f2( self ):
      print( locals() )
      super

x = X()
x.f1()
x.f2()
```

```
{'self': <__main__.X object at 0x7f1773762390>}
{'self': <__main__.X object at 0x7f1773762390>, '__class__': <class '__main__.X'>}
```

  - This value of `__class__` is also available in the child functions.
  - The parser marks up code objects usage of "super". It doesn't have to be a call, it can also be a local variable. If the `super` builtin is assigned to another name and that is used without arguments, it won't work unless `__class__` is taken as a closure variable.
  - As can be seen in the CPython3.2 code, the closure value is added after the class creation is performed.
  - It appears, that only functions locally defined to the class are affected and take the closure.

This left Nuitka with the strange problem, of how to emulate that.

The solution is this:

  - Under Python3, usage of `__class__` as a reference in a function body that is not a class dictionary creation, marks it up via `markAsClassClosureTaker`.
  - Functions that are marked up, will be forced to reference variable to `__class__`.

> ***Note***
>
> This one should be optimized away later if not used. Currently we have "no unused closure variable" detection, but it would cover it.

  - When recognizing calls to `super` without arguments, make the arguments into variable reference to `__class__` and potentially `self` (actually first argument name).
  - Class dictionary definitions are added.

    These are special direct function calls, ready to propagate also "bases" and "metaclass" values, which need to be calculated outside.

The function bodies used for classes will automatically store `__class__` as a shared local variable, if anything uses it. And if it's not assigned by user code, it doesn't show up in the "locals()" used for dictionary creation.

Existing "__class__" local variable values are in fact provided as closure, and overridden with the built class , but they should be used for the closure giving, before the class is finished.

So "__class__" will be local variable of the class body, until the class is built, then it will be the "__class__" itself.

Frame Stack

In Python, every function, class, and module has a frame. It creates created when the scope it entered, and there is a stack of these at run time, which becomes visible in tracebacks in case of exceptions.

The choice of Nuitka is to make this non-static elements of the node tree, that are as such subject to optimization. In cases, where they are not needed, they may be removed.

Consider the following code.

```
def f():
    if someNotRaisingCall():
        return somePotentiallyRaisingCall()
    else:
        return None
```

In this example, the frame is not needed for all the code, because the condition checked wouldn't possibly raise at all. The idea is the make the frame guard explicit and then to move it downwards in the tree, whenever possible.

So we start out with code like this one:

```
def f():
    with frame_guard( "f" ):
        if someNotRaisingCall():
            return somePotentiallyRaisingCall()
        else:
            return None
```

This is to be optimized into:

```
def f():
    if someNotRaisingCall():
        with frame_guard( "f" ):
            return somePotentiallyRaisingCall()
    else:
        return None
```

Notice how the frame guard taking is limited and may be avoided, or in best cases, it might be removed completely. Also this will play a role when inling function, it will not be lost or need any extra care.

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

The `assert` statement is a special statement in Python, allowed by the syntax. It has two forms, with and without a second argument. The later is probably less known, as is the fact that raise statements can have multiple arguments too.

The handling in Nuitka is:

```
assert value
# Absolutely the same as:
if not value:
    raise AssertionError
```

```
assert value, raise_arg
# Absolutely the same as:
if not value:
    raise AssertionError, raise_arg
```

This makes assertions absolutely the same as a raise exception in a conditional statement.

This transformation is performed at tree building already, so Nuitka never knows about `assert` as an element and standard optimizations apply. If e.g. the truth value of the assertion can be predicted, the conditional statement will have the branch statically executed or removed.

The "comparison chain" expressions

```
a < b > c < d
# With "temp variables" and "assignment expressions", absolutely the same as:
a < ( tmp_b = b ) and tmp_b > ( tmp_c = c ) and ( tmp_c < d )
```

This transformation is performed at tree building already. The temporary variables keep the value for the potential read in the same expression. The syntax is not Python, and only pseudo language to expression the internal structure of the node tree after the transformation.

This useful "keeper" variables that enable this transformation and allow to express the short circuit nature of comparison chains by using `and` operations.

The `execfile` builtin

Handling is:

```
execfile( filename )
# Basically the same as:
exec( compile( open( filename ).read() ), filename, "exec" )
```

> **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 `yield`

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:

```
@decorator
def function():
   pass
# Is basically the same as:
def function():
   pass
function = decorator( function )
```

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.

Therefore in Nuitka this assignment is from a "function body expression" and only the last decorator returned value is assigned to the function name.

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

Inplace Assignments

Inplace assignments are 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.

Complex Assignments

Complex assignments are defined as those with multiple targets to assign from a single source and are re-formulated to such using a temporary variable and multiple simple assignments instead.

```
a = b = c
```

```
_tmp = c
b = _tmp
```

```
a = _tmp
del _tmp
```

This is possible, because in Python, if one assignment fails, it can just be interrupted, so in fact, they are sequential, and all that is required is to not calculate `c` twice, which the temporary variable takes care of.

Unpacking Assignments

Unpacking assignments are re-formulated to use temporary variables as well.

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

Becomes this:

```
_tmp = h()

_iter1 = iter( _tmp )
_tmp1 = unpack( _iter1, 3 )
_tmp2 = unpack( _iter1, 3 )
_tmp3 = unpack( _iter1, 3 )
unpack_check( _iter1 )
a = _tmp1
b.attr = _tmp2
c[ind] = _tmp3
d = _tmp
_iter2 = iter( _tmp )
_tmp4 = unpack( _iter2, 3 )
_tmp5 = unpack( _iter2, 3 )
_tmp6 = unpack( _iter2, 3 )
unpack_check( _iter1 )
e = _tmp4
f = _tmp5
g = _tmp6
```

That way, the unpacking is decomposed into multiple simple statementy. It will be the job of optimizations to try and remove unnecessary unpacking, in case e.g. the source is a known tuple or list creation.

Note

The `unpack` is a special node which is a form of `next` that will raise a `ValueError` when it cannot get the next value, rather than a `StopIteration`. The message text contains the number of values to unpack, therefore the integer argument.

Note

The `unpack_check` is a special node that raises a `ValueError` exception if the iterator is not finished, i.e. there are more values to unpack.

With Statements

The `with` statements are re-formulated to use temporary variables as well. The taking and calling of `__enter__` and `__exit__` with arguments, is presented with standard operations instead. The promise to call `__exit__` is fulfilled by `try/except` clause instead.

```
with some_context as x:
    something( x )
```

```
tmp_source = some_context

# Actually it needs to be "special lookup" for Python2.7, so attribute lookup won't
# be exactly what is there.
tmp_exit = tmp_source.__exit__

# This one must be held for the whole with statement, it may be assigned or not, in
# our example it is. If an exception occurs when calling "__enter__", the "__exit__"
# should not be called.
tmp_enter_result = tmp_source.__enter__()

try:
    # Now the assignment is to be done, if there is any name for the manager given,
    # this may become multiple assignment statements and even unpacking ones.
    x = tmp_enter_result

    # Then the code of the "with" block.
    something( x )
except Exception:

    # Note: This part of the code must not set line numbers, which we indicate with
    # special source code references, which we call "internal". Otherwise the line
    # of the frame would get corrupted.

    if not tmp_exit( *sys.exc_info() ):
        raise
else:
    # Call the exit if no exception occurred with all arguments as "None".
    tmp_exit( None, None, None )
```

Note

We don't refer really to `sys.exc_info()` at all, instead, we have references to the current exception type, value and trace, taken directory from the caught exception object on the C++ level.

If we had the ability to optimize `sys.exc_info()` to do that, we could use the same transformation, but right now we don't have it.

For Loops

The for loops use normal assignments and handle the iterator that is implicit in the code explicitely.

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

This is roughly equivalent to the following code:

```
_iter = iter( iterable )
_no_break_indicator = False

while True:
    try:
        _tmp_value = next( _iter )
    except StopIteration:
        # Set the indicator that the else branch may be executed.
        _no_break_indicator = True

        # Optimization should be able to tell that the else branch is run only once.
        break

     # Normal assignment re-formulation applies to this assignment of course.
     x, y = _tmp_value
     del _tmp_value

     if something( x ):
         break

if _no_break_indicator:
    otherwise()
```

> ***Note***
>
> The `_iter` temporary variable is of course in a temp block and the `x, y` assignment is the normal is of course re-formulation of an assignment that cannot fail.
>
> The `try/except` is detected to allow to use a variant of `next` that throws no C++ exception, but instead to use `ITERATOR_NEXT` and which returns NULL in that case, so that the code doesn't really have any Python level exception handling going on.

While Loops

Loops in Nuitka have no condition attached anymore, so while loops are re-formulated like this:

```
while condition:
    something()
```

```
while True:
    if not condition:
        break
```

```
    something()
```

This is to totally remove the specialization of loops, with the condition moved to the loop body in a conditional statement, which contains a break statement.

That makes it clear, that only break statements exit the loop, and allow for optimization to remove always true loop conditions, without concerning code generation about it, and to detect such a situation, consider e.g. endless loops.

> ***Note***
>
> Loop analysis can therefore work on a reduced problem (which breaks are executed under which conditions) and be very general, but it cannot take advantage of the knowledge encoded directly anymore. The fact that the loop body may not be entered at all, if the condition is not met, is something harder to discover.

Exception Handler Values

Exception handlers in Python may assign the caught exception value to a variable in the handler definition.

```
try:
    something()
except Exception as e:
    handle_it()
```

That is equivalent to the following:

```
try:
    something()
except Exception:
    e = sys.exc_info()[1]
    handle_it()
```

Of course, the value of the current exception, use special references for assignments, that access the C++ and don't go via `sys.exc_info` at all, these are called `CaughtExceptionValueRef`.

Statement `try/except` with `else`

Much like `else` branches of loops, an indicator variable is used to indicate the entry into any of the exception handlers.

Therefore, the `else` becomes a real conditional statement in the node tree, checking the indicator variable and guarding the execution of the `else` branch.

Classes Creation

Python2

Classes have a body that only serves to build the class dictionary and is a normal function otherwise. This is expressed with the following re-formulation:

```
# in module "SomeModule"
# ...

class SomeClass( SomeBase, AnotherBase )
    """ This is the class documentation. """

    some_member = 3
```

```
def _makeSomeClass:
    # The module name becomes a normal local variable too.
    __module__ = "SomeModule"

    # The doc string becomes a normal local variable.
    __doc__ = """ This is the class documentation. """

    some_member = 3

    return locals()

    # force locals to be a writable dictionary, will be optimized away, but that
    # property will stick. This is only to express, that locals(), where used will
    # be writable to.
    exec ""

SomeClass = make_class( "SomeClass", (SomeBase, AnotherBase), _makeSomeClass() )
```

That is roughly the same, except that `_makeSomeClass` is *not* visible to its child functions when it comes to closure taking, which we cannot express in Python language at all.

Therefore, class bodies are just special function bodies that create a dictionary for use in class creation. They don't really appear after the tree building stage anymore. The type inference will of course have to become able to understand `make_class` quite well, so it can recognize the created class again.

Python 3

In Python3, classes are a complicated way to write a function call, that can interact with its body. The body starts with a dictionary provided by the metaclass, so that is different, because it can "__prepare__" a non-empty locals for it, which is hidden away in "prepare_class_dict" below.

What's noteworthy, is that this dictionary, could e.g. be a "OrderDict". I am not sure, what "__prepare__" is allowed to return.

```
# in module "SomeModule"
# ...

class SomeClass( SomeBase, AnotherBase, metaclass = SomeMetaClass )
    """ This is the class documentation. """

    some_member = 3
```

```
# Non-keyword arguments, need to be evaluated first.
tmp_bases = ( SomeBase, AnotherBase )

# Keyword arguments go next, __metaclass__ is just one of them. In principle we
```

```
# need to forward the others as well, but this is ignored for the sake of
# brevity.
tmp_metaclass = select_metaclass( tmp_bases, SomeMetaClass  )

tmp_prepared = tmp_metaclass.__prepare__( "SomeClass", tmp_bases )

# The function that creates the class dictionary. Receives temporary variables
# to work with.
def _makeSomeClass:
    # This has effect, currently I don't know how to force that in Python3 syntax,
    # but we will use something that ensures it.
    locals() = tmp_prepared

    # The module name becomes a normal local variable too.
    __module__ = "SomeModule"

    # The doc string becomes a normal local variable.
    __doc__ = """ This is the class documentation. """

    some_member = 3

    # Create the class, share the potential closure variable __class__ with others.
    __class__ = tmp_metaclass( "SomeClass", tmp_bases, locals() )

    return __class__

# Build and assign the class.
SomeClass = _makeSomeClass()
```

List Contractions

TODO.

Set Contractions

TODO.

Dict Contractions

TODO.

Generator Expressions

There are re-formulated as functions.

Generally they are turned into calls of function bodies with (potentially nested) for loops:

```
gen = ( x*2 for x in range(8) if cond() )
```

```
def _gen_helper( __iterator ):
   for x in __iterator:
      if cond():
          yield x*2

gen = _gen_helper( range(8 ) )
```

Boolean expressions and and or

The short circuit operators `or` and `and` tend to be only less general that the `if/else` expressions and are therefore re-formulated as such:

```
expr1() or expr2()
```

```
_tmp if ( _tmp = expr1() ) else expr2()
```

```
expr1() and expr2()
```

```
expr2() if ( _tmp = expr1() ) else expr1()
```

In this form, the differences between these two operators becomes very apparent, the operands are simply switching sides.

With this the branch that the "short-circuit" expresses, becomes obvious, at the expense of having the assignment expression to the temporary variable, that one needs to create anyway.

Simple Calls

As seen below, even complex calls are simple calls. In simple calls of Python there is still some hidden semantic going on, that we expose.

```
func( arg1, arg2, named1 = arg3, named2 = arg4 )
```

On the C-API level there is a tuple and dictionary built. This one is exposed:

```
func( *( arg1, arg2 ), **{ "named1" : arg3, "named2" : arg4 } )
```

A called function will access this tuple and the dictionary to parse the arguments, once that is also re-formulated (argument parsing), it can then lead to simple inlining. This way calls only have 2 arguments with constant semantics, that fits perfectly with the C-API where it is the same, so it is actually easier for code generation.

Although the above looks like a complex call, it actually is not. No checks are needed for the types of the star arguments and it's directly translated to `PyObject_Call`.

Complex Calls

The call operator in Python allows to provide arguments in 4 forms.

- Positional (or normal) arguments
- Named (or keyword) arguments
- Star list arguments
- Star dictionary arguments

The evaluation order is precisely this. An example would be:

```
something( pos1, pos2, name1 = named1, name2 = named2, *star_list, **star_dict )
```

The task here is that first all the arguments are evaluated, left to right, and then they are merged into only two, that is positional and named arguments only. for this, the star list argument and the star dict arguments, are merged with the positional and named arguments.

What's peculiar, is that if both the star list and dict arguments are present, the merging is first done for star dict, and only after that for the star list argument. This makes a difference, because in case of an error, the star argument raises first.

```
something( *1, **2 )
```

This raises "TypeError: something() argument after ** must be a mapping, not int" as opposed to a possibly more expected "TypeError: something() argument after * must be a sequence, not int."

That doesn't matter much though, because the value is to be evaluated first anyway, and the check is only performed afterwards. If the star list argument calculation gives an error, this one is raised before checking the star dict argument.

So, what we do, is we convert complex calls by the way of special functions, which handle the dirty work for us. The optimization is then tasked to do the difficult stuff. Our example becomes this:

```
def _complex_call( called, pos, kw, star_list_arg, star_dict_arg ):
    # Raises errors in case of duplicate arguments or tmp_star_dict not being a
    # mapping.
    tmp_merged_dict = merge_star_dict_arguments( called, tmp_named, mapping_check( called, tmp_star_dict ) )

    # Raises an error if tmp_star_list is not a sequence.
    tmp_pos_merged = merge_pos_arguments( called, tmp_pos, tmp_star_list )

    # On the C-API level, this is what it looks like.
    return called( *tmp_pos_merged, **tmp_merged_dict )

returned = _complex_call(
    called        = something,
    pos           = (pos1, pos2),
    named         = {
        "name1" : named1,
        "name2" = named2
    },
    star_list_arg = star_list,
    star_list_arg = star_dict
)
```

The call to "_complex_call" is be a direct function call with no parameter parsing overhead. And the call in its end, is a special call operation, which relates to the "PyObject_Call" C-API.

Nodes that serve special purposes

Side Effects

When an exception is bound to occur, and this can be determined at compile time, Nuitka will not generate the code the leads to the exception, but directly just raise it. But not in all cases, this is the full thing.

Consider this code:

```
f( a(), 1 / 0 )
```

The second argument will create a `ZeroDivisionError` exception, but before that `a()` must be executed, but the call to `f` will never happen and no code is needed for that, but the name lookup must still succeed. This then leads to code that is internally like this:

```
f( a(), raise ZeroDivisionError )
```

which is then modeled as:

```
side_effect( a(), f, raise ZeroDivisionError )
```

where you can consider side_effect a function that returns the last expression. Of course, if this is not part of another expression, but close to statement level, side effects, can be converted to multiple statements simply.

Another use case, is that the value of an expression can be predicted, but that the language still requires things to happen, consider this:

```
a = len( ( f(), g() ) )
```

We can tell that `a` will be 2, but the call to `f` and `g` must still be performed, so it becomes:

```
a = side_effects( f(), g(), 2 )
```

Modelling side effects explicitely has the advantage of recognizing them easily and allowing to drop the call to the tuple building and checking its length, only to release it.

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 pre-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, but not mix and use declarations from existing header code.
2. Allowance: May use `ctypes` module at compile time to ask things about `ctypes` and its types.
3. Goal: Should make use of `ctypes`, to e.g. not hard code what `ctypes.c_int()` gives on the current platform, 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 Discussion

Main 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. This is called "Aliasing".

When that value is a compile time constant, we will want to push it forward, because storing such a constant under a variable name has a cost and loading it back from 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 even more. We want to forward propagate abstract properties of the values.

> ***Note***
>
> Builtin exceptions, and built-in names are also compile time constants.

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, e.g. `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 built-in functions already, where we use real built-ins where possible to make computations. We have the problem though that built-ins may have problems to execute everything with reasonable compile time cost.

Another example, consider the following code:

```
len( "a" * 1000000000000 )
```

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 huge constant and stream it, the C++ code would become 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 `10000000000000`.
- Can predict every of its elements when subscripted, sliced, etc., if need be, with a function we may create.

Similar is true for this horrible thing:

```
range( 10000000000000 )
```

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

- The result is a `list` or `PyListObject`
- We know its length exactly, `10000000000000`
- 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. Therefore Nuitka avoids executing these calculation, when they result in constants larger than a treshold of 256. It's also applied to integers and more CPU and memory traps.

Now lets look at a use case:

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

Looking at this example, one traditional way to look at it, would be to turn `range` into `xrange`, note that `x` is unused. That would already perform better. But really better is to notice that `range()` generated values are not 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 of having an iteration source. Being able to 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 at run time 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" * 1000000000000 )
```

The theme here, is that when we can't compute all intermediate expressions, and we sure can't do it 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.

> ***Note***
>
> The optimization of `len` has been implemented and works for all kinds of container building and ranges.

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 an assignment from a `__import__` expression to the variable `ctypes`. It can be predicted by default to be a module object, and even better, it can be 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 will 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. 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 correct, 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 this means to functions and their call boundaries, 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 "unknown PyObject parameter object", then something that definitely has an `append` attribute, when returned. Otherwise an exception occurs. The type of `a` changes to that after `a.append` look-up succeeds. It might be many kinds of an object, but e.g. it could have a higher probability of being a `PyListObject`. And we would know it cannot be a `PyStringObject`, as that one has no "append".

> ***Note***
>
> If classes, i.e. other types 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` succeeded, 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] # Could be decided now
```

Goal: The structure we use makes it easy to tell what `my_append` may be. So, there should be a means to ask it about call results with given type/value information. We need to be able to tell, if evaluating `my_append` makes sense with given parameters or not, if it does impact the return value.

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

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

The exactness of statements may vary. But some things may be more interesting. If e.g. the aliasing of a parameter value is known exactly, then information about it need to all be given up, but can survive.

It would be nice, if `my_append` had sufficient information, so we could specialize with `list` and `int` from the parameters, and then e.g. know at least some things that it does in that case. Such specialization would have to be decided if it makes sense. In the alternative, it could be done for each variant anyway, as there won't be that many of them.

Doing this "forward" analysis 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 loops (both "for" and "while") has its own problem. The loop start and may have an assumption from before it started, that "a" is constant, but that is only true for the first iteration. So, we can't pass knowledge from outside loop forward directly into the for loop body.

So while we pass through the loop, we need to collect in-validations of this outside knowledge. The assignment to "a" should make it an alternative to 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 in-validations.

For a start, it will be done like this though. At loop entry, all knowledge is removed about everything, and so is at loop exit. That way, only the loop inner working is optimized, and before and after the loop as separate things. The optimal handling of "a" in the example code will take a while.

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 we need to have the constraint collection know when it enters a conditional branch, and then it does, it must take special precautions, to merge the existing state at condition exit. When exiting both 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.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. With that knowledge the `append` attribute call will become the `list` built-in type operation.

- 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.

- Branches that abort make a difference.

```
if type( a ) is list:
    a.append( x )
else:
    raise ValueError
```

Here it is obvious, that the conditional statement exit, requires no merging. We can fully inherit the state of the non-exiting branch, including the knowledge that `a` is in fact a `list` built-in object.

Excursion to `return` statements

The return statement (like `break`, `continue`, `raise`) is "aborting" to control flow. It is always the last statement of inspected block.

If all branches of a conditional statement are "aborting", the statement is decided "aborting" too. If a loop doesn't break, it is "aborting" too.

> ***Note***
>
> The removal of statements following "aborting" statements is implemented, and so is the discovery of abortative conditional statements. It's not yet done for loops, temp blocks, etc. though.

So, return statements are easy for local optimization. In the general picture, it would be sweet to collect all return statements, and analyze the commonality of them. This would give us the "my_append" information from above. And were we to do this for exception raises too, we could tell exceptions from a function too.

Excursion to `yield` expressions

The `yield` expression can be treated like a normal function call, and as such invalidates some known constraints just as much as they do. It executes outside code for an unknown amount of time, and then returns, with little about the outside world known anymore.

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 Nuitka ought to be able to make no difference in performance for it.

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 "tri-state".

  Part of the interface is a method `computeNode` which gives the node the chance to return another node instead, which may also be an exception.

The `computeNode` may be able to produce exceptions or constants even for non-constant inputs depending on the operation being performed. For every expression it will be executed in the order in which the program control flow goes for a function or module.

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.

By default, attribute lookup, should turn an expression to unknown, unless something in the registry can say something about it. That way, `some_list.append` produces something which when called, invalidates `some_list`, but only then.

- 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 or next to the node that produces them.

- 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.
    - `onAssignment( variable, 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 done in a specialized optimization "value propagation" and can be used to implement optimizations in a consistent and fast way. It walks the tree and asks each node to compute. When it encounters assignments, it asks for value friends that can be queries for arguments, and these can be used for the builtins own "computeNode" or value friend decisions.

Note

Assignments to attributes, indexes, slices, etc. will also need to follow the flow of "append", so it cannot escape attention that a list may be modified. Usages of "append" that we cannot be sure about, must be traced to exist, and disallow the list to be considered known value again.

Code Generation Impact

Right now, code generation assumes that everything is a "PyObject *", i.e. 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 and a node may offer it for itself (constant reference is an obvious example) or may delegate the task to an instantiated object of "ValueFriendBase" inheritance. This will e.g. be done, if a state is attached, e.g. the current iteration value.

Goal 1

Initially most things will only be able to give up on about anything. And it will be little more than a tool to do simple lookups in a general form. It will then be the first goal to turn the following code into better performing one:

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

to:

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

and then:

```
a = 3
b = 7
c = 0
return c
```

and then:

```
a = 3
b = 7
c = 0
return 0
```

> ***Note***
>
> This is implemented, but not active for releases, because it's not yet safe, because we are missing detections for mutable values, which later goals will give.

The assignments to "a", "b", and "c" shall become prey to "unused" assignment analysis in the next step. 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.

```
return 0
```

Goal 2

It appears, that "dead value analysis" for "a" and "b" requires that we trace to the end of the scope, if a variable value is or might become used.

For that, we trace the last assignment of each variable, or a new assignment, or "del" statement on it, we decide, if the original assignment to the name was needed or not. If the value wasn't used, but it did provide a reference, we remove the name from it. If it didn't provide a reference, we can make it an expression only.

That would, starting with:

```
3
7
0
return 0
```

give us:

```
return 0
```

which is the perfect result.

In order to be able to manipulate statements that made assignments to names later on, we need to track the exact node(s) that did it. It may be multiple in case of conditions.

```
if cond():
    x = 1
elif other():
    x = 3

# Not using "x".
return 0
```

In the above case, the merge of the value friends, should say that "x" may be undefined, or one of "1" or "3", but since "x" is not used, apply the "dead value" trick to each branch.

> ***Note***
>
> This is totally unimplemented.

Goal 3

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 in 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 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 have a limited history only and be mutated as the processing goes.
- 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 an 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.

- 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.

- Make "SELECT_METACLASS" meta class selection transparent.

  Looking at the "SELECT_METACLASS" it should become an anonymous helper function. In that way, the optimization process can remove choices at compile time, and e.g. inline the effect of a meta class, if it is known.

  This of course makes most sense, if we have the optimizations in place that will allow this to actually happen.

- Accesses to list constants sometimes chould become tuple constants.

```
for x in [ 1, 2, 7 ]:
   something( x )
```

  Should be optimized into this:

```
for x in ( 1, 2, 7 ):
   something( x )
```

  Otherwise, code generation suffers from assuming the list may be mutated and is making a copy before using it. Instead, it would be needed to track, if that list becomes writable, and if it's used as a list.

```
# Examples, where lists need to be maintained, even if not written to
print [ 1,2 ]
print type( [ 1,2 ] )
```

  The best approach is probably to track down `in` and other potential users, that don't use the list nature and just convert then.

- Terminal assignments without effect removal.

  In order to optimize away unused assignments, Nuitka should not try and find variables that are only assigned. It should instead for each assignment find the uses of the value. Two cases then

  1. No more read use before next assignment or end of scope.

     Can remove the assignment nature and make it instead a temp variable of the scope, if the release has an impact (will "__del__" have an effect?).

  2. Value is read.

     Keep it.

- Friends that keep track

  The value friends should become the place, where variables or values track their use state. The iterator should keep track of the "next()" calls made to it, so it can tell which value to given in that

case.

And then there is a destroy, once a value is released, which could then make the iterator decide to tell its references, that they can be considered to have no effect, or if they must not be released yet.

That would solve the "iteration of constants" as a side effect and it would allow to tell that they can be removed.

That would mean to go back in the tree and modify it long after.

```
a = iter( ( 2, 3 ) )
b = next( a )
b = next( a )
del a
```

It would be sweet if we could recognize that:

```
a = iter( ( 2, 3 ) )
b = side_effect( next( a ), 2 )
b = side_effect( next( a ), 3 )
del a
```

That trivially becomes:

```
a = iter( ( 2, 3 ) )
next( a )
b = 2
next( a )
b = 3
del a
```

When the "del a" is happening (potentially end of scope, or another assignment to it), we would have to know of the "next" uses, and retrofit that information that they had no effect.

```
a = iter( ( 2, 3 ) )
b = 2
b = 3
del a
```

- Friends that link

```
a = iter( ( 2, 3 ) )
b = next( a )
b = next( a )
del a
```

When "a" is assigned, it is receiving a value friend, "fresh iterator", for the unused iterator, one that hasn't be used at all.

Then when next() is called on "a" value, it creates *another* value friend, and changes the value friend in the collection for "a" to "used iterator 1 time". It is very important to make a copy.

It is then asked for a value friend to be assigned to "b". It can tell which value that would be, but it has to record, that before "a" can be used, it would have to execute a "next" on it. This is delaying

that action until we see if it's necessary at all. We know it cannot fail, because the value friend said so.

This repeats and again a new "value friend" is created, this time "used iterator 2 times", which is asked for a value friend too. It will keep record of the need to execute next 2 times (which we may have optimized code for).

```
a = iter( ( 2, 3 ) )
b = 2
# Remember a has one delayed iteration
b = 3
# Remember b has two delayed iteration
del a
```

When then "a" is deleted, it's being told "onReleased". The value friend will then decide through the value friend state "used iterator 2 times", that it may drop them.

```
a = iter( ( 2, 3 ) )
b = 2
b = 3
del a
```

Then next round, "a" is assigned the "fresh iterator" again, which remains in that state and at the time "del" is called, the "onReleased" may decide that the assignment to "a", bearing no side effects, may be dropped. If there was a previous state of "a", it will move up.

Also, and earlier, when "b" is assigned second time, the "onReleased" for the constant, bearing no side effects, may also be dropped. Had it a side effect, it would become an expression only.

```
a = iter( ( f(), g() ) )
b = next( a )
b = next( a )
del a
```

```
a = iter( ( f(), g() ) )
b = f()
b = g()
del a
```

```
f()
b = g()
```

That may actually be workable. Difficult point, is how to maintain the trace. It seems that per variable, a history of states is needed, where that history connects value friends to nodes.

```
a = iter(
  (
     f(),
     g()
  )
)
# 1. For the assignment, ask right hand side, for computation. Enter computeNode for
```

```
# iterator making, and decide that it gives a fresh iterator value, with a known
# "iterated" value.
# 2. Link the "a" assignment to the assignment node.
b = next( a )
# 1. ask the right hand side, for computation. Enter computeNode for next iterator
# value, which will look up a.
b = next( a )
del a
```

- Aliasing

  Each time an assignment is made, an alias is created. A value may have different names.

  ```
  a = iter( range(9 ))
  b = a
  c = next(b)
  d = next(a)
  ```

  If we fail to detect the aliasing nature, we will calculate "d" wrongly. We may incref and decref values to trace it.

  To trace aliasing and non-aliasing of values, it is a log(n**2) quadratic problem, that we should address efficiently. For most things, it will happen that we fail to know if an alias exists. In such cases, we will have to be pessimistic, and let go of knowledge we thought we had.

  If e.g. "x" is a list (read mutable value), and aliases to a module value "y", then if we call unknown code, that may modify "y", we must assume that "x" is modified as well.

  For an "x" that is a str (read non-mutable value), aliases are no concern at all, as they can't change "x". So we can trust it rather.

  The knowledge if "x" is mutable or not, is therefore important for preserving knowledge, and of course, if external code, may access aliases or not.

  To solve the issue, we should not only have "variables" in constraint collections, but also "aliases". Where for each variable, module, or local, we track the aliasing. Of course, such an alias can be broken by a new assignment. So, the "variable" would still be the key, but the value would be list of other variables, and then a value, that all of these hold. That list could be a shared set for ease of updating.

  Values produce friends. Then they are assigned names, and can be referenced. When they are assigned names, they should have a special value friend that can handle the alias. They need to create links and destroy them, when something else is assigned.

  When done properly, it ought to handle code like this one.

  ```
  def f():
     a = [ 3 ]
     b = a
     a.append( 4 )
     a = 3
     return b[1]
  ```

  For assignment of "a", the value friend of the list creation is taken, and then it is stored under variable "a". That is already done with an "alias" structure, with only the variable "a". Then when assigning to "b", it is assigned the same value friend and another link is created to variable "b". Then, when looking up "a.append", that shared value is looked up and potentially mutated.

If it doesn't get the meaning of ".append", it will discard the knowledge of both "a" and "b", but still know that they alias.

The aliasing is only broken when a is assigned to a new value. And when then "b" is subscribed, it may understand what that value is or not.

- Value Life Time Analysis

  A value may be assigned, or consumed directly. When consumed directly, it's life ends immediately, and that's one thing. When assigned, it doesn't do that, but when the last reference goes away, which may happen when the name is used for another value.

  In the mean time, the value may be exposed through attribute lookup, call, etc. which may modify what we can tell about it. An unknown usage must mark it as "exists, maybe" and no more knowledge.

- Shelve for caching

  If we ever came to the conclusion to want and cache complex results of analysis, we could do so with the shelve module. We would have to implement "__deepcopy__" and then could store in there optimized node structures from start values after parsing.

- Tail recursion optimization.

  Functions that return the results of calls, can be optimized. The Stackless Python does it already.

- Integrate with "upx" compression.

  Calling "upx" on the created binaries, would be easy.

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.rst

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