Contents

Usage

Requirements

- C++ Compiler: You need a compiler with support for C++03

  Currently this means, you need to use the GNU g++ compiler of at least version 4.4 on Linux or the clang 3.0 compiler on MacOS X, or else the compilation may fail.

  On Windows the MinGW g++ compiler is known to work start with at least version 4.5, the VC++ compiler is not known at this time.

- Python: Version 2.6, 2.7 or 3.2

  You need at least CPython to execute Nuitka and then also to execute the created binary, because it uses the shared library of CPython.

Environment

No changes are required, you can call the "nuitka" and "nuitka-python" executables directly without any changes to the environment. For convinience, it might be easier to add the directory with them to the PATH variable:

Linux/MacOS X/MSYS shell:

```
eval `misc/create-environment`
```

With some luck this also works:

```
. misc/create-environment
```

Windows: Extend "PATH" with the directory containing Nuitka executables. Either have MinGW installed to "C:MinGW" (then Nuitka will find and use it automatically) or also add it to the PATH environment.

Command Line

Nuitka has a "--help" option to output what it can do:

```
nuitka --help
```

The "nuitka-python" command is "nuitka", but with different defaults and tries to compile and directly execute a script:

```
nuitka-python --help
```

These options with different defaults are "--exe" and "--execute", so it is somewhat similar to what plain "python" will do. Note: In the future, the intention is to support CPython's "python" command lines in a compatible way, but currently it isn't so.

Use Case 1 - Program compilation with all modules embedded

If you want to compile a whole program recursively, and not only the single file that is the main program, do it like this:

```
nuitka-python --recurse-all program.py
```

> ***Note***
>
> The is more fine grained control than "--recurse-all" available. Consider the output of "nuitka-python --help".

In case you have a plugin directory, i.e. one which is not found by recursing after normal import statements (recommended way), you can always require that a given directory shall also be included in the executable:

```
nuitka-python --recurse-all --recurse-directory=plugin_dir program.py
```

> ***Note***
>
> If you don't do any dynamic imports, setting your "PYTHONPATH" at compilation time will be sufficient for all your needs normally. Use "--recurse-directory" only if you make "__import__()" calls that Nuitka cannot predict, because they e.g. depend on command line parameters.

> ***Note***
>
> The resulting binary still depends on Python and used C extension modules being installed. Sorry about that, it's not yet a "py2exe" replacement. Please come and help to add that functionality if you would like to see it in Nuitka.

Use Case 3 - Extension Module compilation

If you want to compile a single extension module, all you have to do is this:

```
nuitka some_module.py
```

The resulting "some_module.so" can then be used instead of "some_module.py".

Use Case 3 - Package compilation with all modules embedded

If you need to compile a whole package and embedded all modules, that is also feasible, use it like this:

```
nuitka some_package --recurse-directory=some_package
```

> ***Note***
>
> The recursion into the package directory needs to be provided manually, otherwise the package is empty. Data files located inside the package will not be embedded yet.

Where to go next

Remember, this project is not completed yet. Although the CPython test suite works near perfect, there is still more work needed, to make it do enough optimizations to be worth while. Try it out.

Subscribe to its mailing lists

Please visit the mailing list page in order to subscribe the relatively low volume mailing list. All Nuitka issues can be discussed there.

Report issues or bugs

Should you encounter and issues or bugs, please visit the Nuitka bug tracker and report them.

Contact me via email with your questions

You are welcome to contact me via email with your questions.

Word of Warning

Consider using this software with caution. Your feedback and patches are very welcome.

Especially report it please, if you find that anything doesn't work, because the project is now at the stage that this should not happen.

Join Nuitka

You are more than welcome to join Nuitka development and help to complete the project in all minor and major ways.

The development of Nuitka occurs in git. We currently have these 2 branches:

- master:

  This branch contains the stable release to which only hotfixes for bugs will be done. It is supposed to work at all times and is supported.

- develop:

  This branch contains the ongoing development. It may at times contain little regressions, but also new features. On this branch the integration work is done, whereas new features might be developed on feature branches.

> ***Note***
>
> I accept patch files, git formated patch queues, and git pull requests. I will do the integration work. If you base your work on "master" or "develop" at any given time, I will do any re-basing required and keep your authorship intact.

> ***Note***
>
> The Developer Manual explains the coding rules, branching model used, with feature branches and hotfix releases, the Nuitka design and much more. Consider reading it to become a contributor. This document is intended for Nuitka users.

Unsupported functionality

General

The "co_code" attribute of code objects

The code objects are empty for for native compiled functions. There is no bytecode with Nuitka's compiled function objects, so there is no way to provide bytecode.

Threading can block it seems

Bug tracker link: "Threading is not supported, never yields the execution to other threads"

The generated code never lets the CPython run time switch threads, so its chances to do so are reduced, which may lead to dead lock problems.

Help is welcome to add support for threading to Nuitka.

Start of function call vs. end of function call in traceback output

Bug tracker link: "In tracebacks Nuitka uses start of call line, whereas CPython uses end of call line"

In CPython the traceback points to the end of the function call, whereas in Nuitka they point to the first line of the function call.

This is due to the use of the "ast.parse" over bytecode it seems and not easy to overcome. It would require parsing the Python source on our own and search for the end of the function call.

Maybe someone will do it someday. Help is welcome.

We can consider making the compatible behaviour optional, and use it for the tests only as the called expression clearly is more useful to see then the closing brace.

Optimization

Constant Folding

The most important form of optimization is the constant folding. This is when an operation can be predicted. Currently Nuitka does these for some builtins (but not all yet), and it does it for binary/unary operations and comparisons.

Constants currently recognized:

```
5 + 6     # operations
5 < 6     # comparisons
range(3)  # builtins
```

Literals are the one obvious source of constants, but also most likely other optimization steps like constant propagation or function inlining will be. So this one should not be underestimated and a very important step of successful optimizations. Every option to produce a constant may impact the generated code quality a lot.

Status: The folding of constants is considered implemented, but it might be incomplete. Please report it as a bug when you find an operation in Nuitka that has only constants are input and is not folded.

Constant Propagation

At the core of optimizations there is an attempt to determine values of variables at run time and predictions of assignments. It determines if their inputs are constants or of similar values. An expression, e.g. a module variable access, an expensive operation, may be constant across the module of the function scope and then there needs to be none, or no repeated module variable look-up.

Consider e.g. the module attribute `__name__` which likely is only ever read, so its value could be predicted to a constant string known at compile time. This can then be used as input to the constant folding.

```
if __name__ == "__main__":
   # Your test code might be here
   use_something_not_use_by_program()
```

From modules attributes, only "__name__" is currently actually optimized. Also possible would be at least "__doc__".

Also builtins exception name references are optimized if they are uses as module level read only variables:

```
try:
   something()
except ValueError: # The ValueError is a slow global name lookup normally.
   pass
```

Builtin Call Prediction

For builtin calls like `type`, `len`, or `range` it is often possible to predict the result at compile time, esp. for constant inputs the resulting value often can be precomputed by Nuitka. It can simply determine the result or the raised exception and replace the builtin call with it allowing for more constant folding or code path folding.

```
type( "string" ) # predictable result, builtin type str.
len( [ 1, 2 ] )  # predictable result
range( 3, 9, 2 ) # predictable result
```

```
range( 3, 9, 0 ) # predictable exception, range hates that 0.
```

The builtin call prediction is considered implemented. We can simply during compile time emulate the call and use its result or raised exception. But we may not cover all the builtins there are yet.

Sometimes builtins should not be predicted when the result is big. A "range()" call e.g. may give too big values to include the result in the binary. Then it is not done.

```
range( 100000 ) # We do not want this one to be expanded
```

Status: This is considered mostly implemented. Please file bugs for built-ins that are predictable but are not computed by Nuitka at compile time.

Conditional Statement Prediction

For conditional statements, some branches may not ever be taken, because of the conditions being possible to predict. In these cases, the branch not taken and the condition check is removed.

This can typically predict code like this:

```
if __name__ == "__main__":
   # Your test code might be here
   use_something_not_use_by_program()
```

or

```
if False:
   # Your deactivated code might be here
```

It will also benefit from constant propagations, or enable them because once some branches have been removed, other things may become more predictable, so this can trigger other optimization to become possible.

Every branch removed makes optimization more likely. With some code branches removed, access patterns may be more friendly. Imagine e.g. that a function is only called in a removed branch. It may be possible to remove it entirely, and that may have other consequences too.

Status: This is considered implemented, but for the maximum benefit, more constants needs to be determined at compile time.

Exception Propagation

For exceptions that are determined at compile time, there is an expression that will simply do raise the exception. These can be propagated, collecting potentially "side effects", i.e. parts of expressions that must still be executed.

Consider the following code:

```
print side_effect_having() + (1 / 0)
print something_else()
```

The (1 / 0) can be predicted to raise a "ZeroDivisionError" exception, which will be propagated through the "+" operation. That part is just Constant Propagation as normal.

The call to "side_effect_having" will have to be retained though, but the print statement, can be turned into an explicit raise. The statement sequence can then be aborted and as such the "something_else" call needs no code generation or consideration anymore.

To that end, Nuitka works with a special node that raises an exception and has so called "side_effects" children, yet can be used in generated code as an expression.

Status: The propagation of exceptions is implemented on a very basic level. It works, but exceptions will not propagate through all different expression and statement types. As work progresses or examples arise, the coverage will be extended.

Exception Scope Reduction

Consider the following code:

```
try:
    b = 8
    print range( 3, b, 0 )
    print "Will not be executed"
except ValueError, e:
    print e
```

The try block is bigger than it needs to be. The statement "b = 8" cannot cause a "ValueError" to be raised. As such it can be moved to outside the try without any risk.

```
b = 8
try:
    print range( 3, b, 0 )
    print "Will not be executed"
except ValueError, e:
    print e
```

Status: Not yet done yet. The infrastructure is in place, but until exception block inlining works perfectly, there is not much of a point.

Exception Block Inlining

With the exception propagation it is then possible to transform this code:

```
try:
    b = 8
    print range( 3, b, 0 )
    print "Will not be executed"
except ValueError, e:
    print e
```

```
try:
    raise ValueError, "range() step argument must not be zero"
except ValueError, e:
    print e
```

Which then can be reduced by avoiding the raise and catch of the exception, making it:

e = ValueError("range() step argument must not be zero") print e

Status: This is not implemented yet.

Empty branch removal

For loops and conditional statements that contain only code without effect, it should be possible to remove the whole construct:

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

The loop could be removed, at maximum it should be considered an assignment of variable "i" to 999 and no more.

Another example:

```
if side_effect_free:
   pass
```

The condition should be removed in this case, as its evaluation is not needed. It may be difficult to predict that side_effect_free has no side effects, but many times this might be possible.

Status: This is not implemented yet.

Unpacking Prediction

When the length of the right hand side of an assignment to a sequence can be predicted, the unpacking can be replaced with multiple assignments.

```
a, b, c = 1, side_effect_free(), 3
```

```
a = 1
b = side_effect_free()
c = 3
```

This is of course only really safe if the left hand side cannot raise an exception while building the assignment targets.

We do this now, but only for constants, because we currently have no ability to predict if an expression can raise an exception or not.

Status: Not really implemented, and should use "mayHaveSideEffect()" to be actually good at things.

Builtin Type Inference

When a construct like "in xrange()" or "in range()" is used, it is possible to know what the iteration does and represent that, so that iterator users can use that instead.

I consider that:

```
for i in xrange(1000):
    something(i)
```

could translate "xrange(1000)" into an object of a special class that does the integer looping more efficiently. In case "i" is only assigned from there, this could be a nice case for a dedicated class.

Status: Future work, not even started.

Quicker function calls

Functions are structured so that their parameter parsing and "tp_call" interface is separate from the actual function code. This way the call can be optimized away. One problem is that the evaluation order can differ.

```
def f( a, b, c ):
    return a, b, c

f( c = get1(), b = get2(), a = get3() )
```

This will evaluate first get1(), then get2() and then get3() and then make the call.

In C++ whatever way the signature is written, its order is fixed.

Therefore it will be necessary to have a staging of the parameters before making the actual call, to avoid an re-ordering of the calls to get1(), get2() and get3().

To solve this, we may have to create wrapper functions that allow different order of parameters to C++.

Status: Not even started.

Credits

Contributors to Nuitka

Thanks go to these individuals for their much valued contributions to Nuitka. Contributors have the license to use Nuitka for their own code even if Closed Source.

The order is sorted by time.

- Li Xuan Ji: Contributed patches for general portability issue and enhancements to the environment variable settings.
- Nicolas Dumazet: Found and fixed reference counting issues, import work, improved some of the English and generally made good code contributions all over the place, code generation TODOs, tree building cleanups, core stuff.
- Khalid Abu Bakr: Submitted patches for his work to support MinGW and Windows, debugged the issues, and helped me to get cross compile with MinGW from Linux to Windows. This was quite a difficult stuff.
- Liu Zhenhai: Submitted patches for Windows support, making the inline Scons copy actually work on Windows as well. Also reported import related bugs, and generally helped me make the Windows port more usable through his testing and information.
- Christopher Tott: Submitted patches for Windows, and general as well as structural cleanups, he is also attempting to support direct "ctypes" calls to be evaluated at compile time.
- Pete Hunt: Submitted patches for MacOS X support.

Projects used by Nuitka

The CPython project http://www.python.org/

Thanks for giving us CPython, which is the base of Nuitka.

The gcc project http://gcc.gnu.org/

Thanks for not only the best compiler suite, but also thanks for supporting C++11 which helped to get Nuitka off the ground. Your compiler was the first usable for Nuitka and with little effort.

The Scons project http://www.scons.org/

Thanks for tackling the difficult points and providing a Python environment to make the build results. This is such a perfect fit to Nuitka and a dependency that will likely remain.

The valgrind project http://valgrind.org/

Luckily we can use Valgrind to determine if something is an actual improvement without the noise. And it's also helpful to determine what's actually happening when comparing.

The MinGW project http://www.mingw.org/

Thanks for porting the best compiler to Windows. This allows portability of Nuitka with relatively little effort.

The mingw-cross-env project http://mingw-cross-env.nongnu.org

Thanks for enabling us to easily setup a cross compiler for my Debian that will produce working Windows binaries.

The wine project http://www.winehq.org/

Thanks for enabling us to run the cross compiled binaries without have to maintain a windows installation at all.

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=README.txt

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