GccPy - GCC Front-End for Python

Philip Herron
http://redbrain.co.uk
redbrain@gcc.gnu.org
herron.philip@googlemail.com

April 5, 2011

Abstract

Gccpy is a new front-end to Gnu Compiler Collection which implements the very popular multi-paridigm dynamic language Python. In here i discuss the ideas, techniques and aproachs taken to implment this as a staticaly compiled language.

Creating statically compiled languages has been generally aimed for more 'low-level' languages such as C/C++/Fortran where the language requires strong typing and other kinds declarative features; which brings about to some degree much less dynamic features which languages like Python/PHP/Perl take for granted. The reason these more 'high-level' languages are able to do such things is due to the fact they are generaly implemented as interpreted languages and this allows for much of this dynamic logic to take place at runtime when a program is passed through their respective interpreters.

Contents

List of Figures

Listings

Listings

1 Preface

GCCPY is an attempt at creating a Staticly compiled version of Python 2.6 using GCC as a framework for middle-end, back-end optimization aswell as protable code-generation. Creating statically compiled languages has been generally aimed for more 'low-level' languages such as C/C++/Fortran where the language requires strong typing and other kinds declarative features; which brings about to some degree much less dynamic features which languages like Python/PHP/Perl take for granted. The reason these more 'high-level' languages are able to do such things is due to the fact they are generaly

implemented as interpreted languages and this allows for much of this dynamic logic to take place at runtime when a program is passed through their respective interpreters.

1.1 Acknowledgements

y Family, Andi Hellmund, Ian Lance Taylor, Paul Biggar, GCC irc.oftc.net community, GCC, GCC-help, Linux Outlaws, Jezra, Nybill, Windigo, yaMatt, Fabsh, MethodDan, ... Belfast Linux User Group, comp.compilers, help-bison, help-flex, . If it wasnt for Paul Biggars Phd work into PHC the php compiler it probably wouldnt have given me the inspiration to try building this project. FINISH..

2 Introduction

GCCPY is an attempt at creating a Staticly compiled version of Python 2.6 using GCC as a framework for middle-end, back-end optimization aswell as protable code-generation. Creating statically compiled languages has been generally aimed for more 'low-level' languages such as C/C++/Fortran where the language requires strong typing and other kinds declarative features; which brings about to some degree much less dynamic features which languages like Python/PHP/Perl take for granted. The reason these more 'high-level' languages are able to do such things is due to the fact they are generaly implemented as interpreted languages and this allows for much of this dynamic logic to take place at runtime when a program is passed through their respective interpreters.

2.1 Approaches and Techniques

2.1.1 Interpreters

nterpreters generaly work via several main ways, a very stright forward first principles approach will lean towards creating a parser for the language which creates AST's into DAG structures which the language runtime will:

```
symbol * o = foo;
while (o) {
      eval_symbol (o);
      o = o->next;
}

// Simple DAG style struct to represent tree's
struct symbol {
       enum sym_Type T;
       union { ... } opa;
       union { ... } opb;
}
```

Or something similar so you can loop over a symbol table and evaluate each stmt at a time as the parser passes this to the middle-end.

And things like garbage collection will run as things are evaluated a simple way garbage collection is handled in this kind of approach for an interpreter is by the notion of contexts this is at least in my opinion a very first principle and stright forward approach to it.

Take for example a language like Perl where you might create a functions such as:

```
my $global_var = 55;

sub my_routine {
  $x = 1;
  $y = 2;

  return $x + $y + $global_var;
}

sub main {
    x = &my_routine ()
}

&main ()
```

So following this statment of code line by line and how it could be evaluated, the first stmt is a simple assignment but any identifiers in a dynamicaly typed language are nothing but lables for a dictionary.

So to implement this you can create an object in memory which holds the data that the value is an integer of value 55; Which is pointed to by a dictionary by hash'd value of global_var.

Then the method or subrountine of my_routine is stored in the same dictionary with the hash'd value of my_routine, the same for the main rotuntine. And finally the method main is called this is where things get interesting and the notion of contexts come into play.

If we consider the lexical scope of these identifiers, we have a toplevel area where the identifiers:

```
{ "global_var", "my_rountine", "main" }
```

are initilized with respective values in memory, then what happens this lexical scope when a function is called, another context is created, ie another dictionary is pushed onto the stack is a nice way of implementing this lexical scoping in such a way we can have some clever assumptions when builing a garbage collector.

So when we go into this new context we are inside the main subroutine so the runtime stack would look something like this:

```
 ----------------
 | x           | -> result of my_routine -> ref_count = 1
 ----------------
 | global_var  | -> 55                   -> ref_count = 1
 | my_routine  | -> subroutine           -> ref_count = 1
 | main        | -> subroutine           -> ref_count = 1
 ----------------
```

So when the runtime wants to lookup the value of an identifier it will start off in the HEAD of the stack then go down the stack of dictionaries untill it finds the value it might look for.

So far this is pretty stright forward and i havent shown the link how this idea makes clever assumptions possible in a garbage collector. When inside the main routine we call the my_routine, routine.

The lexical scope isnt as simple as pushing another dictionary onto the stack since thats not how the language works. So when the language is called a new stack frame is generated for this so when we are evaluating symbols inside this new context it looks like:

```
 ---------------
| x             | -> 1                                -> ref_count = 1
| y             | -> 2                                -> ref_count = 1
| T.1           | -> y + global_var = 55 + 2 = 57 -> ref_count = 1
| T.2           | -> x + T.1         = 57 + 1 = 58 -> ref_count = 1
 ---------------          (will be 2 since its assigned in the return)
 ---------------
| global_var    | -> 55                     -> ref_count = 1  -------
| my_routine    | -> subroutine             -> ref_count = 1   | x | -> T.2
| main          | -> subroutine             -> ref_count = 1  -------
 ---------------
```

The context frame when we are inside the main subroutine is just pop'd off the stack before we enter, the my_routine function to preserve the language lexical scope. But once we return from a function we can immedietly decrement the ref_count on each object that was created in that context.

The interesting part is because we return the object T.2 which in turn increments its ref_count to 2, but as i stated we pop out of that context and everything is decremented so when the garbage collector run's over the runtime will be able to pick up on this.

In reality it works very similar to how something like an i386 architecture has things in memory for example if we wanted something like this in C code:

```
int foobar (void)
{
  int x = 1, y = 2;
  return x + y;
}
```

We could generate an IR of:

```
int foobar (void)
{
  int x, y, T.1;
  x = 1;
  y = 2;
  T.1 = x + y;
  return T.1;
}
```

And the i386 code of:

```
.globl foobar
foobar:
  subl $12, %esp        # get some stack space
  mov $1, %esp          # x
  mov $2, 4(%esp)       # y
```

```
mov 4(%esp), 8(%esp) # setting up a very
                          # highlevel/*un-optimized* addition
addl %esp, 8(%esp)   # T.1
mov 8(%esp), %eax    # the return
addl $12, %esp       # fix the stack
ret
```

This is where some interpreters/runtimes start to try and become much more like a 'virtual machine' like Java they implement their language by having a runtime which runs code that is in a virtual inscrution set. So when they parse their language with a front-end they generate this virtual instruction set for the given program but then they 'compile/assemble' this to bytecode which is a similar akin to C where we assemble the target code to an object code before linking into an executeable format. But really the byte code is nothing more than a binary form of the instruction set to optimize execution of the instruction set.

2.1.2 Virtual Machines

The difference with a virtual instruction to say for example an i386 or other hardware instruction set is that it will have many many highlevel nmemonics and instructions available to the programmer such as declaring subroutines with mutators like public or protected etc. And more nmenoics for even type conversion i2b/d2i object creation and manipulation new/putfield, stack management swap/dup2 and many more which hardware instruction sets dont have.

This adds some benifits to the language runtime in that there is room for many more optimiztions in the code generation, which in turn should streamline the code such that it should run faster than this previous style of evaluation tree's which can be quite memory intensive, quite slow and unoptimized, but gives a very streamlined way of implementing prototype languages quickly for proof of concept. Though to both of these methods new techniques are being implemented like Jit'ing (Just in Time compilation) to further optimize the code, a well known and highly used implementation of that technique is the Java Hot Spot.

2.1.3 Static Compilation

The last method is the method with which i have chosen to build Gccpy a python implmentation ontop of GCC, it is actualy a very under used and under aprectiated way of building languages recently even dynamic ones like python as it tends to only be considered for low level languages with strong typing and more rigid boundaries with which the programmer will have to follow to allow for efficient code generation, since many of the features that make things dynamic are eliminated. What actually happens is the front-end parses the code and generates target code for a cpu which you can assemble, link and execute. So you just generate code directly to be run on the OS instead of a virtual one where it is still run on a runtime envoirnment.

The benifit in doing this method is you cut out a huge layer of code in the middle which controls many many structures in memory and the control of the virtual cpu. Even the code and the sheer complexity of adding a Jit to a runtime the code to manage its use can still factor in to some degree some preformance degredation.

Where as the staticaly compilated method has everything pre generated and pre-jit'd in a point of view. The problem is finding a balance where the sheer complexity of allowing much of the dynamic features to be handled efficiently without relying on the runtime library of the language for huge amounts of work. Even elimitinating the dicitionary lookup with interpreters gives huge preformance increases since generating code can point to locations on the stack directly instead of having lookup.

From here i will discuss the thought process and design that i have decided to follow in building this staticly compiled version of Python. Python is a very popular and highlevel language at the moment given rapid development cycles and many highlevel features allowing for many abstract and complex transformations or declarations to be achieved through very little work; this has become the key challenge in designing gccpy, allowing for all these powerful dynamic features yet creating efficient output so the end result is still a new viable implementation of the language.

3 Gccpy

Firstly we need to look at the archiecture of GCC and how and why i choose this to be a platform for building this implementation. Gcc has been around for many many years and has become the defacto standard of C compilers on unix with little competition untill recent developments with LLVM and some of the propriotary implementations from Sun/Oracle and Intel suite of compilers, and many lesser known ones for much more specific roles for hardware requirements or outputs. Because of GCC's success its had many eyes over the years, constant development and usage that its has generaly become very good at what its key roles have been in creating efficient code on many many architectures. It follows a very standard design but crucialy probably the first of its kind of revolve around such a rigid style that become so well used thoughout the industry.

If we consider a programmer creating some code, he must save his code to a source file we then pass this to the compiler though some invokation in this case as:

```
gcc <some other options> -c -o bla.c bla.o
```

And we will be left with an assembled object code file of that input code. But lets analyse the steps GCC takes to achive this. A standard compiler such as GCC will follow this style:

```
Front-end -> Middle-end -> Back-end
```

3.1 Front-end

Starting with the Front-end we have in GCC it split into two parts, we have the compiler-driver and the compiler-proper. The compiler driver is the program the programmer or user will invoke to compile their code like gcc/g++/gfortran/etc... here the program will analyse the options the user passed to the program and setup apropriate things to invoke the compiler proper program which will do all the work and generate the target asm code. After that is completed depending on the options passed the compiler driver will invoke an assembler and linker to finilize the program for the user to run their code.

So all the work is completed in the compiler proper, here we start with the front-end as i stated which is responsible with reading and parsing the input code into structures we can work with, in GCC's case we have 2 options. GENERIC and GIMPLE which are both valid IR's you can use but mostly front-ends will use GENERIC which abstracts many parts of the middle-end and gives a more highlevel language to represent structures and constructs of a language or can be translated to this at least, it could/believed to be difficult to represent something functional like Haskell or Lisp within these IR but i will get to that later and why i still belive its more than valid to do so.

3.1.1 Static Analysis

An example why many people belive generating efficient code for dynamic languages can be difficult is take for example:

```
def foo (x):
    x.append (1)
    return x + [ 1, 2, 3 ]
```

So what happens here in an abstract point of view you can't assume anything about this code due to dynamic typing compared to something like

```
def List foo (List x):
    x.append (1)
    return x + [ 1, 2, 3 ]
```

Having storage specifiers insantly makes this set of code much more declaritive and gives rise to many more assumptions able to be made; which in turn gives a compiler more 'hints' on what it can do to generate code. Of course in the example above this is just a hypothetical language just to demonstrate the idea. So to implement dynamic typing we have to analyse what it actually is.

Lets take a normal/regular python session:

```
>>> x = 1
>>> x = "string"
>>> y = x
>>> x = 2
```

So what is actually happening now line by line by showing what each identifier is assigned to what data.

```
>>> x = 1                  # x = 1          | y = NULL
>>> x = "string"           # x = "string"   | y = NULL
>>> y = x                  # x = "string"   | y = "string"
>>> x = 2                  # x = 2          | y = "string"
```

So why is this actually a problem, traditionly take for example code like:

```
int x = 1
x = 1.5555
x = "string"
```

When a c-compiler would run over that code it would give all manar of warnings about type conversion and invalid types being assigned. But why is this since a compiler will want to generate efficient code it will reserve the space valid for an integer on the stack which on an i386 32bit processor would be 32 bits usualy and would a use subl $4, %esp to have space on the stack for that integer, but the problem arises if we were to then want to put in data which is greater than the size previously allocated for the given initial data. So you will have overflow and corruption of data. So how can you combat that to make dynamic typing work, the method or approach i have taken for gccpy takes much inspiration in how object orientation works. Every piece of static data give in a program is wraped into a gpy_object_t structure at runtime so in turn every type in gccpy is implemented via a gpy_object_t type, so for example the previous python session could be represented in GIMPLE via something like:

```
gpy_object_t * x = fold_integer (1)
incr_ref_count (x)

decr_ref_count (x)
x = fold_string ("string")

gpy_object_t * y = x
incr_ref_count (y)

x = fold_integer (2)
incr_ref_count (x)
```

3.1.2 Dynamic Typing

The basic idea how dynamic typing is not what an identifier with a specified storage specifier holds its what an identifier points to. So when

```
x = fold_integer (1)
```

We should look at what the gpy_object_t structure looks like currently its in many ways similar to how PY_object works in the cpython implemetation but is a little more specific and streamlined to gccpy's needs.

```
typedef struct gpy_object_t {
  enum GPY_OBJECT_T T;
  union{
    gpy_object_state_t * object_state;
    struct gpy_callable__t * call;
    gpy_literal_t * literal;
  } o ;
} gpy_object_t ;
```

This structure is quite open to be used in many areas of how gccpy works but what we are interested in is the:

```
gpy_object_state_t * object_state;
```

This is the part where it stores the static data defined be it an integer or a class defined in the source code.

```
typedef struct gpy_rr_object_state_t {
  char * obj_t_ident;
  signed long ref_count;
  void * self;
  struct gpy_typedef_t * definition;
} gpy_object_state_t ;
```

This structure is whats used to hold the object_state it holds the object type identifier as a string the reference count for the garbage collector the pointer to a structure in memory which is the actual data for example int or FILE etc, and also a pointer to the objects definition structure. Each object has its own definition and each definition requires several hooks:

```
typedef struct gpy_typedef_t {
  char * identifier;
  size_t builtin_type_size;
  gpy_object_t * (*init_hook)(struct gpy_typedef_t *, gpy_object_t **);
  void (*destroy_hook)(gpy_object_t *);
  void (*print_hook)(gpy_object_t * , FILE *, bool);
  struct gpy_number_prot_t * binary_protocol;
  struct gpy_builtin_method_def_t * methods;
} gpy_typedef_t ;
```

It has the identifier the size of the sturcture of which holds the actual data the initilization hook which returns the object state when you initilize an object the destroy hook for the garbage collector, a print hook for the print keyword to print the data. Now things get more interesting, the binary protocol is whats used to allow for dynamic binary operators so things like:

```
x = 2 + 1.5

concat = "foo" + "bar"
```

Can allow for mixed type binary operations, by having hooks for each type of binary operation be it addition subtraction etc. Finaly there is a table of member methods which allows for dot accesors like:

```
list.append (...)
list.index (...)
```

As append and index are both member methods to the builtin type List. Taking one step back from the runtime library lets look at how GCC fits in. As before we started to discuss the architecture of GCC Front-end the compiler proper, we have have our parser which so far has been a Flex and GNU/Bison implementation of the python 2.6 syntax designed by Andi Hellmund; From this parser it was clear it was nessecary to introduce a new form of IR to our front-end as well as using GERNERIC and gcc's middle-end to bring this down to GIMPLE from the rest of the compilation process.

3.2 Py_dot

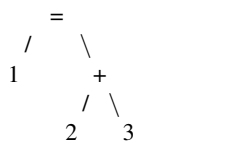
This new IR is called PY_dot, this is a similar design to how GccGO implmented using gogo by Ian Lance Taylor which has also been a key element in helping this project take shape.

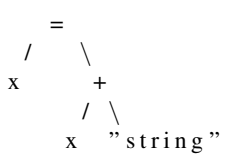
PY_dot is a very highlevel IR which is mostly just DAG's which just represents the constructs the parser tells us. This is where gccpy has to get clever, as i stated before taking a slice of python code in an abstract acedmic point of view you would say you cannot assume anything about the code as it is parsed. But we have to look at examples more closely to find patterns of how things work within the python language in a more practical way.

So lets take for example a pice of simple code:

```
x = 1 + 2 + 3
x += "string"
```

The compiler cannot assume that X is of type Integer even though all the operands will be of type int, because a next line will mess that up because its operand is of type String or const char *. Taking from basic compiler front-end theory we construct a DAG of the first expression and the next.





This is what Py_dot represents for us, so from here we have to do quite a bit of work to analyse where we can declare our data and evaluate it; since we still have to bring this down to a form with which the target system can compute.

So what we need to do is come back to that idea of contexts firstly, since everything we need to do we need to declare it in GENERIC much like C where we need to declare a variable before we use it. As soon as we see this first assignment we need to lookup in the current context if 'x' was declared or not and if not we declare the object such that this block of code would generate the IR

```
gpy_object_t * x, T.1, F.1, F.2, F.3, F.4;

F.1 = fold_int (3)
F.2 = fold_int (2)

T.1 = eval_bin_op (F.2, F.1, OP_ADDITION)

F.3 = fold_int (1)
x = eval_bin_op (F.1, T.1, OP_ADDITION)
incr_ref_count (x)

F.4 = fold_string ("string")
decr_ref_count (x)
x = eval_bin_op(x, F.4, OP_ADDITION)
incr_ref_count (x)
```

So this illustates the approach we can take to implement dynamic typing since the stack and lexical scope of names/identifiers are simple but there are still more complex constructs or cases to look at. Firstly lets look at how python works by a simple example:

```
x = 2              (1)
y = "string"       (2)
def foobar (x):    (3)
    return x^2     (4)
print foobar (4)   (5)
```

So difference from the previous example that block of stmts scoping can be achieved by standard c-decl stack usage example on i386 would be:

```
subl \$24, \%esp # gets all 24 bytes of space on the stack
                 # for gpy_object_t * x, T.1, F.1, F.2, F.3, F.4

And before we leave the code block with the ret instruction we can simply:

addl \$24, \%esp
```

To fix the stack again and lexical scoping is preserved. But because languages like C require code to be wrapped inside functions that makes it simple unlike python. In python execution of code starts immedietly we can simply start computing data as we want. So to illustrate why this previous code sample requires another another approach to allow for correct scoping we will examine the stack as it should be executed.

After execution of (1):

```
_____
| x | -> 2
_____
```

After 2

```
_____
| x | -> 2
| y | -> "string"
_____
```

After 3/4

```
____________
| x        | -> 2
| y        | -> "string"
| foobar   | -> callable
____________
```

After

3.3 Developing the Data Model

Since Python is at its core an object orientated model, taking inspiration from C++ and how its data model is implmented we can tackle this using classes. So for example lets take a look at some basic python code and translate it into an object orientated data model.

```
# main.py
x = 1
y = 2
def foobar () :
  return x+y
print foobar ()
```

Could be translated into:

```
class __mangled_modulemain_PY
{
  gpy_object_t * x , y;

  gpy_object_t * foobar (gpy_object_t **)
  {
    gpy_object_t * T.1 = eval_expr (x, y, OP_ADDITION)
    return T.1;
  }

  void module_initilizer (void)
  {
    x = fold_int (1)
    y = fold_int (2)

    T.2 = foobar (NULL)
    eval_print (1, T.2)
  }
}
```

```
int main (int argc, char *argv[])
{
  init_runtime ();

  class __mangled_modulemain_PY program;
  program->module_initilizer ();

  finalize_runtime ();

  return 0;
}
```

3.3.1 Handling Object Orientation

3.4 Control Structures

3.4.1 Conditionals

3.4.2 Loops

4 Python's Dynamic Features

5 References

5.1 Links

5.2 Index

References

[1] Lolcode Team, http://lolcode.com/