UCPy: Reverse-Engineering Python

John Aycock David Pereira Georges Jodoin
Department of Computer Science
University of Calgary
Calgary, Alberta, Canada T2N 1N4
{aycock|pereira|jodoin}@cpsc.ucalgary.ca

1 Introduction

One of the recurring topics in the Python community is how to make Python programs run faster. Typically, a set of solutions is proposed which include: adding static type inference; somehow compiling programs into native code; translating Python programs into Parrot/Lisp/.net code; applying research results from dynamically-typed language implementation. Progress has been made on some of these, such as Psyco [8], but many of these proposed solutions are qualified by the caveat *no one has the time/resources to work on it.*

In the Programming Languages Lab at the University of Calgary we have a research project underway, UCPy, whose short-term goal is to examine ways we can make Python run faster. We have learned some lessons through our design and implementation work to date, about both Python and the undertaking of such a project, which we present in this paper.

2 Design of UCPy

> 'If you have four groups working on a compiler, you'll get a 4-pass compiler.' – statement of Conway's Law [2]

In deference to the above quote, UCPy is divided into three parts: a compiler, an assembler/linker, and a virtual machine. This division was in part for nontechnical reasons – compiler implementation was initially proceeding in parallel with virtual machine design – but also had some practical benefits. The compiler was much easier to debug with a human-readable assembly language output, and the virtual machine had been tested using assembly inputs prior to the compiler emitting code.

Currently, the individual parts are not combined into one integrated executable unit, so Python's `exec` and `eval` statements (which require the compiler to be present at run time) are not supported yet.

2.1 The Python Compiler: `ucpc`

`Ucpc` is written in C, using the standard compiler tools flex and bison [4]. This is in contrast to the CPython compiler, which uses a weaker type of parser generated by a parser generator distributed with Python. This difference alone has led to some interesting discoveries about the Python grammar, which we discuss in the next section.

The compiler constructs a representation of the whole module being compiled as an abstract syntax tree (AST). Each AST node has at most two children, and this binary representation makes it amenable for use with the tree pattern-matching tool burg [3], by which we find a tree covering and generate code for the AST.

Eventually, we intend to turn `ucpc` into a retargetable compiler, and have a back end that generates code for the CPython virtual machine too.

2.2 The Assembler/Linker: `pyas`

The combined assembler and linker is written in Python, using the SPARK toolkit [1]. It consists of just under 1100 lines of code, about 23% of which is for emitting the binary format that the virtual machine requires.

The assembly output from `ucpc` is essentially a collection of objects, such as code objects, strings, integers, and floats, some of which have references to one another. For example, a code object may make references to a number of string and integer objects. `Pyas`, as it assembles the code, determines the dependencies between objects so it may link them accordingly; unused objects are discarded.

As an assembler, `pyas` performs a number of bookkeeping tasks such as calculating the contents of tables for exception handling and mapping virtual machine addresses into source line numbers, which greatly simplify the task of the compiler.

2.3 The Virtual Machine: `mamba`

Our new virtual machine for Python is hand-written in C, to allow fair timing comparisons between it and CPython. It is a register-based virtual machine as opposed to the stack-based architecture used in CPython, and a minimal one at that: `mamba` has 19 instructions compared to CPython's 103 instructions. The full architectural details of `mamba` are outside the scope of this paper, but are described fully in [6].

`Mamba` is intended as a framework for optimization experiments, and has features to facilitate this, most notably a generic framework for garbage collection. Currently, we have four different collectors which may be chosen when the virtual machine is compiled: a "null" collector which never reclaims garbage; a reference-counting collector; a mark-sweep collector; a collector with a new algorithm that dynamically infers memory regions [5].

We expect to have enough functionality in `mamba` to allow running a full suite of microbenchmarks within the next few months.

3 Reverse-Engineering Lessons

Reverse-engineering an established programming language is a major endeavor, even if the language were *not* rapidly evolving. However, Python is in a constant state of flux, and we have had to set limits on the feature set just to make the scope of UCPy achievable. We are targeting the functionality of Python 1.5.2 officially, at least in the virtual machine, although the compiler is capable of handling most Python constructs up to version 2.x. Unofficially, the "creeping feature creature" has been hard at work, and `mamba` accepts rich comparisons and list comprehensions among other things.

A particular problem has been the vast number of built-in modules and functions that Python has. The "batteries included" philosophy of Python is a labor-intensive one to duplicate when building a new system from scratch, and we have consequently adopted a lazy approach – in the technical sense of the term – to adding necessary functionality.

In the remainder of this paper, we present details of what we have learned about the nooks and crannies of Python as a result of building UCPy, including a look at Python's design from a minimalist re-implementation point of view.

3.1 Lessons from Grammar School

In our compiler, we use bison to generate the parser. The form of grammar that bison uses as input is equivalent to, but slightly different from, the grammar specification allowed by Python's parser generator tool.[1] We therefore had to convert the grammar from one form to the other.

Bison's checking of grammars is more stringent than Python's parser generator, however, and having performed the grammar conversion we found two ambiguities in the Python grammar. Ambiguities are a problem in programming languages, just as they are in natural languages like English, because users expect their program to have a unique interpretation. We verified each ambiguity in the original grammar, to ensure that we had not made an error in translation.

The first ambiguity was with the following two grammar rules:

```
factor: ('+'|'-'|'~') factor | power
power: atom trailer* ('**' factor)*
```

Essentially, a `power` can end with zero or more instances of a `**` followed by a `factor`. However, a `factor` can itself be a `power`. This means that, for input such as

```
123 ** 123 ** 123
```

there are two ways to interpret this, according to the grammar. By changing the grammar rules, so that the `**` and `factor` were optional but could not be repeated, fixed this ambiguity:

[1] Specifically, bison doesn't allow EBNF constructs.

```
factor: ('+'|'-'|'~') factor | power
power: atom trailer* ['**' factor]
```

It is interesting to note that the grammar in the reference manual was correct, but the implementation was not [7].

The second ambiguity was stranger. Inside a pair of backquotes in Python, any expression list should be legal, including the program:

```
a = 123
`a,`
```

The trailing comma in the expression `a,` should create a tuple, to be converted into a string by the backquotes. The Python interpreter soundly rejected the program, though. Upon inspection, the Python grammar has an ambiguity here as well. Basically, it's analogous to having a grammar for arithmetic expressions, where both sides of nested expressions are denoted using left parentheses – you can't tell when you've seen the end of one expression versus the start of a new, nested expression.

What we find curious about this second ambiguity is that, despite it being visible to the programmer, it had clearly been sitting in the Python interpreter for some time. Perhaps Python programmers don't use backquotes!

3.2 Lots and Lots of Batteries Included

The Python libraries are extremely large. The library modules written in Python aren't a problem – presumably when we have a full Python implementation, the library modules written in Python will just work. However, there is a large body of built-in modules written in C too. In Python 1.5.2, there were over 59,000 lines of C code in these modules; in Python 2.2.2, there are over 71,000 lines. Both these numbers exclude comments and blank lines.

There is no reason to expect that the size and complexity of the built-in modules is going to decrease in future releases of Python. In terms of reverse-engineering Python, creating a full reimplementation of these modules is daunting, not to mention the effort of ongoing maintenance. There are three basic strategies we could adopt for implementing built-in modules:

1. Reuse. If we retained Python's C API, conceivably we could make use of the existing module code. That poses a large constraint on the degree to which we can introduce radical changes into UCPy, though.

2. Rewrite (in Python). This is probably the fastest and most flexible route, but it would automatically place UCPy at a severe disadvantage in timing comparisons with CPython. Given that our short-term goal is to improve upon Python's speed, this is not the best choice.

3. Rewrite (in C). We have chosen this labor-intensive path to implementing built-in module functionality. As we discover the need for parts of built-in modules (typically based on what specific benchmark programs require), we implement those parts.

Rewriting modules in C also means writing code that uses the garbage collection interface properly, a not-always-trivial task. We are exploring ways to simplify this process considerably.

3.3 RISC vs. CISC: A Reprise

Since our short-term goal is to make Python programs run faster, one obvious avenue is to look at ways to optimize virtual machine code and the means of executing the code.

With its large number of instructions, the CPython VM may be thought of as a "CISC-VM," by way of analogy to real CPU architectures. In our experience, it is difficult to reason about such a large VM in terms of code optimization, and any nontrivial VM experiments are expensive in terms of implementation time.

`Mamba`, in contrast, can be considered a "RISC-VM." It has a small instruction count and, apart from where Python's semantics dictate otherwise, the instructions are quite simple. It also features a register-based instruction set, which adds to the instruction size but makes dependencies explicit – a useful feature for code optimization.

Although each `mamba` instruction is four bytes long, most of that space encodes either register numbers or offsets. There are few bits remaining for the instruction type itself, so there is relatively little room for expansion. This means that a RISC-VM's instruction set has to be gotten right initially; it is not necessarily easy to add in new instructions at a later date. All of the Python language, not just a subset, had to be encompassed by our instruction set design.

A critical factor in the performance of RISC architectures is the level of compiler sophistication. We are too early in the development of UCPy to ascertain if this holds true for RISC-VM architectures as well.

3.4 "The Reference Manual says *what*?"

The Python language and CPython have grown up together, and CPython has gradually evolved to support Python's features. In contrast, designing a RISC-VM from scratch was tremendously challenging. Clean designs had to be repeatedly discarded because they were unable to support obscure details of Python. The most troublesome spots are described below.

3.4.1 Bound Methods

When `mamba` was first designed, we wanted to investigate an alternative model of bound method objects. For example, if one views a function as having a curried form (that is, not as accepting a tuple of objects but instead as accepting just the first parameter and returning a curried function on the remaining parameters), then bound objects can be viewed as a restriction of currying to just the first parameter. Currying would simulate bound method objects correctly, and add a very useful and expressive feature to the Python language.

Although a design with currying was considered, it was abandoned because of the difficulty of integrating it with the exotic features of the function Python function call interface: */** parameters and arguments, default-valued parameters, tuplified and keyword arguments.

Some Python code relies explicitly on the presence of bound method objects, and the freedom to inspect and modify the fields within them. If we had removed bound method objects in favor of curried functions we would have broken these Python programs. Also, function calls that would normally fail if not enough parameters were passed would succeed in curried form. Currying would represent a major semantic change to Python and was therefore not implemented.

3.4.2 Function and Method Calls

The Python function call interface is the amalgamation of many different mechanisms: */** parameters and arguments, default-valued and tuplified parameters, keyword arguments. The logic to implement the call instruction in the virtual machine is by far the most complicated of all the instructions in the machine.

One of the main problems encountered concerned tuple parameters:

```
def a((b, c)):
        pass
```

We opted for a simple solution – emit code in the function's prologue to deconstruct the given tuple argument and then assign each component of the tuple to the corresponding parameter name in the local namespace. Conceptually, this would be like treating the above function as if it were written:

```
def a(_tmp):
        b = _tmp[0]
        c = _tmp[1]
        pass
```

An advantage of this approach is that this work need not be done by the `call` instruction. The consequences are that the unpacking of tuple parameter arguments is slower since it is executed in virtual machine code instead of in native code. Also, if there is an error such as an insufficient number of components in the argument tuple, then an error is raised *by the unpacking code* which is one activation record level deeper than the caller, which is where the CPython virtual machine would raise the exception.

3.4.3 Coercion

The design of our virtual machine is based on the transformation of statements `A op B` into the form `A.op(B)`, where `A` and `B` are objects. For example, `a[i]` is transformed into `a.__getitem__(i)`. Ideally, we want to compile `a + b` into

`a.__add__(b)`, thereby eliminating the need for an `OP_ADD` instruction and other arithmetic instructions.

However, the arithmetic semantics of Python do not make this idea viable. Even if it were possible to factor out the coercion logic into a `COERCE` instruction, niceties such as reversed operations would still be problematic – in the above example, if object `a` does not have an `__add__` operation, an `AttributeError` exception will be raised. Python semantics then require that a search for an `__radd__` function occur. Implementing this logic in VM code would require that the instruction which retrieves the function attribute `__add__` be surrounded by an exception handler *and* exception handling code which searches for an `__radd__` method. Not only would this lead to code bloat, given the widespread use of arithmetic instructions, but it would also result in a performance penalty.

In our virtual machine, the solution was to add an `OP` instruction which contains the appropriate logic to handle incremental operations, coercions, normal, and reflected operations.

3.4.4 Exceptions

Consider the following code:

```
class C1: pass
class C2: pass
class C3: pass

def f1(): return C1
def f2(): return C2
def f3(): return C3

try:
        raise C3
except f1():
        pass
except f2():
        pass
except f3():
        pass
```

Here, in order to determine which exception handler handles the exception, the expressions `f1()`, `f2()`, and `f3()` have to be evaluated at run-time. Even the types of the `except` expressions are not known at compile-time. As a result, each expression must be evaluated and compared to the value of the exception until an exception is found. Even worse is the fact that the execution of one `except` expression could potentially change the value of a following `except` expression, thereby imposing a top to bottom order of evaluation upon them. There is no easy way to jump directly to the exception handler as there is in other languages.

3.4.5 TMTOWTDI, a.k.a. Fallbacks

As mentioned, methods such as `__add__` fall back to methods such as `__radd__`. An important observation is that the fallback mechanism requires operands to be *changed* prior to being passed to the fallback function. In the case of `__radd__` the operands are interchanged. For arithmetic, the logic to interchange the operands is built directly into our `OP` instruction.

Comparison operations also require special treatment because of their use of fallbacks. A comparison such as `a < b` cannot simply be compiled as `a.__lt__(b)` since, in the absence of the `__lt__` method, a fallback to the `__cmp__` method is required. Although the virtual machine can simply call `a.__cmp__(b)` instead of `a.__lt__(b)`, the return values are different: if `a < b`, `__lt__` will return 1, but `__cmp__` will return -1. As a result, the virtual machine instruction which extracts methods from objects must provide a wrapper for the extracted comparison method which translates the return values appropriately.

4 Summary

UCPy is a new implementation of Python, which we are using for research to improve the performance of dynamically-typed languages. Exploring the dark corners of Python has revealed a number of difficult issues. To fix these outright would result in a language which is not Python, but perhaps the problems can be smoothed over as Python continues to evolve.

5 Acknowledgments

This work was supported in part by the National Science and Engineering Research Council of Canada. Thanks to Guido van Rossum and Tim Peters for responding quickly to our bug reports.

References

[1] J. Aycock. Compiling little languages in Python. In *Proceedings of the 7th International Python Conference*, pages 69–77, 1998.

[2] E. Raymond (editor). Jargon file 4.3.3. http://tuxedo.org/~esr/jargon, September 2002.

[3] C. W. Fraser, R. R. Henry, and T. A. Proebsting. Burg – fast optimal instruction selection and tree parsing. *ACM SIGPLAN Notices*, 27(4):68–76, 1992.

[4] J. R. Levine, T. Mason, and D. Brown. *Lex & Yacc*. O'Reilly & Associates, second edition, 1992.

[5] D. Pereira and J. Aycock. Dynamic region inference. Technical Report 2002–709–12, University of Calgary, Department of Computer Science, 2002.

[6] D. Pereira and J. Aycock. Instruction set architecture of Mamba, a new virtual machine for Python. Technical Report 2002–706–09, University of Calgary, Department of Computer Science, 2002.

[7] T. Peters. Personal communication, May 2002.

[8] A. Rigo. Psyco (specializing compiler for Python). http://psyco.-sourceforge.net.