pylibjit: A JIT Compiler Library for Python*†

Gergö Barany
Institute of Computer Languages
Vienna University of Technology
gergo@complang.tuwien.ac.at

Abstract: We present pylibjit, a Python library for generating machine code at load time or run time. The library can be used for dynamic code generation, although it is mainly aimed at optimizing compilation of existing Python function definitions.

Python functions can be compiled by simply annotating them with a decorator specifying data types. The pylibjit compiler takes advantage of this type information to perform unboxing, omit dynamic type checks, and avoid unnecessary reference counting operations.

We discuss pylibjit's compilation schemes based on an abstract syntax tree or Python's internal bytecode representation, and present benchmark results. For simple numerical programs, we achieve speedups of up to 50 × over standard interpreted Python.

1 Motivation

Interpreted high-level programming languages are often criticized for their comparatively poor performance. Still, they are popular due to their ease of use and high-level features. Various techniques are used to bridge the performance gap to more traditional compiled languages: tracing just-in-time compilers (JITs) [CSR+09, BCFR09], JITs for subsets of languages marked by special annotations [asm], or ahead-of-time compilers that rely on static type annotations [Sel09, num] or type inference [she]. A partial solution is to use interpreters that specialize the program at run time [Bru10].

This paper introduces the `pylibjit` library which aims to provide a JIT for fragments of Python code marked by annotations ('decorators' in Python parlance). The compiled code runs within the normal Python interpreter embedded in a traditional Python program; program parts without compiler annotations are interpreted as usual, while the compiled parts can make use of static type information provided by the developer.

This is in contrast to most other Python compilers, such as PyPy [BCFR09]. PyPy is a complete re-implementation of the Python language and uses an object model that is incompatible with the standard Python interpreter (known as CPython). This means that if

*Extended version of a paper that appeared as preprint at Kolloquium Programmiersprachen (KPS), Lutherstadt Wittenberg, October 2013.

```
def fib(n):
    if n < 2:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

Figure 1: The naïve Fibonacci function in Python.

a Python project is to use PyPy, *all* of the project must be runnable in PyPy; however, PyPy cannot be used with many popular Python libraries which have parts written against the interpreter's C API. As a popular example, the popular Qt GUI library is not compatible with PyPy. Such problems do not occur with `pylibjit`, which the programmer only applies to a few selected functions, while the rest of the program runs in the interpreter as before. Further, `pylibjit` is compatible with CPython's API, so even Python functions manipulating objects of externally defined types can be compiled. Compared to PyPy, however, `pylibjit` is currently built on a much less aggressively optimizing compilation scheme.

JITs are mostly associated with languages that are compiled to some form of 'bytecode' intermediate representation. However, machine code generation at run time is an important topic even for some programs written in traditional, statically compiled languages. Code generation can be useful for programs that perform expensive computations with control flow that heavily depends on user input. For example, image processing applications can benefit from generating specialized JIT-ed code for user-defined image filters [Pet07]. Even the Linux kernel contains a JIT compiler for a simple domain-specific language describing network packet filtering rules [Cor11].

Our `pylibjit` project builds on a JIT library meant for building code at run time in such applications. However, since Python also allows execution of code at load time (or rather, since there is no clear distinction between 'run time' and 'load time'), we can apply the same library to compile entire Python function definitions as they appear in a source file, while the file is being loaded at program startup. The following sections describe how `pylibjit` builds up from a simple JIT library interface to what is a de facto ahead-of-time compiler for a large subset of the Python language.

2 Compiling functions using the low-level API

Our `pylibjit` library builds on the GNU `libjit` just-in-time compiler library[1] and an existing Python wrapper library for it[2]. We will use the naïve Fibonacci function in Figure 1 to illustrate the use of this existing library and our improvements to it. The library exports a class `jit.Function` which users must subclass for each function they wish

[1] http://www.gnu.org/software/libjit/

[2] https://github.com/eponymous/libjit-python

```
class fib_function(jit.Function):
    def create_signature(self):
        # argument order: return type, argument types
        return self.signature_helper(jit.Type.int, jit.Type.int)

    def build(self):
        # values: n, one, two
        n = self.get_param(0)
        one = self.new_constant(1, jit.Type.int)
        two = self.new_constant(2, jit.Type.int)
        # if n < 2: goto return_label
        return_label = self.new_label()
        self.insn_branch_if(n < two, return_label)
        # return fib(n-one) + fib(n-2)
        fib_func = self
        fib_sig = self.create_signature()
        a = self.insn_call('fib', fib_func, fib_sig, [n-one])
        b = self.insn_call('fib', fib_func, fib_sig, [n-two])
        self.insn_return(a + b)
        # return_label: return n
        self.insn_label(return_label)
        self.insn_return(n)
```

Figure 2: Building the Fibonacci function using the low-level JIT interface.

---

to build and compile. Figure 2 shows the code needed to build the Fibonacci function using this interface. The code defines a class `fib_function` inheriting from `jit.Function` containing two methods `create_signature` and `build`. Single-line comments are introduced using the # sign.

The core is the `build` function which performs the API calls to build up the intermediate code step by step: API calls provide for computations, labels, conditional and unconditional jumps, and function calls. Note that, since the intermediate code is meant to be compiled to machine code, all computations are typed, although type annotations are mostly only needed where a value is first introduced; the types of arithmetic and other operators are inferred from their operands. The type name `int` refers to the machine's 'usual' word-sized integer type, as in C.

Note also that already at this level, Python's high-level nature offers some convenience: The arithmetic operators + and − (and others) are overloaded to work on the compiler's 'value' objects, so we can directly generate an addition instruction by writing `a + b` rather than the less natural `insn_add(a, b)`.

Having created this definition, the class can be instantiated and the instance called as if it were a normal Python function. Wrapper code takes care of unboxing Python argument values and boxing the native code's return value in a Python object.

3 The higher-level API

While this library is usable, it makes building functions more verbose than absolutely necessary. Having to subclass `Function` and implement several methods on it can lead to a lot of boilerplate code; the only interesting function in a typical subclass is `build`, so ideally users should not have to write more than this function. Conveniently, Python provides a concept of *function decorators* that can be used for this purpose.

A decorator is a callable object that can be attached to a function definition using the `@` operator. After the function has been parsed and compiled to Python bytecode, the decorator is applied to it and can perform any analysis or transformation. Finally, the decorator's return value replaces the original function object.

That is, the effect of a decorator `example_decorator` on a function definition:

```
@example_decorator
def func():
    ...
```

is equivalent to defining the function, then calling the decorator manually and using its value to replace the defined object:

```
def func():
    ...
func = example_decorator(func)
```

This enables various higher-order programming techniques based on wrapping or bypassing function calls.

Using this mechanism, it is easy to write a decorator for JIT builder functions that hides all the boilerplate involved in defining a class and instantiating it. This is encapsulated in the `jit.builder` decorator exported by `pylibjit`. This decorator creates an internal class subclassing `jit.Function`, defines the `build` method in that class to call the decorated function provided by the user, and finally takes care of instantiating the JIT-ed function.

Besides boilerplate, another inconvenience when using the pure `libjit` interface is the lack of structure. In Figure 2, the control flow in the generated program is difficult to see as it is implicit in a number of labels and jump statements. We can, however, build constructs for more structured programs using Python's *context managers* combined with its `with` statement. In essence, a context manager is a pair of two functions `__entry__` and `__exit__`. A statement of the form:

```
with foo:
    # statements
    ...
```

wraps the enclosed block of statements in a context manager `foo` (which must be defined elsewhere). Whenever execution reaches the `with` statement, first the context man-

```
@jit.builder(return_type=jit.Type.int,
             argument_types=[jit.Type.int])
def fib2(func):
    n = func.get_param(0)
    one = func.new_constant(1, jit.Type.int)
    two = func.new_constant(2, jit.Type.int)
    with func.branch(n < two) as (false_label, end_label):
        func.insn_return(n)
    # else:
        func.insn_label(false_label)
        func.insn_return(func.recursive_call('fib', [n - one]) +
                         func.recursive_call('fib', [n - two]))
```

Figure 3: Building the Fibonacci function using the higher-level interface.

ager's `__entry__` function is called. Any values returned by `__entry__` may be bound to variables using the `as` keyword. Afterwards, the block is executed as usual. Whenever execution of the block ends in any way, including by returns or exceptions, the context manager's `__exit__` function is called before execution or stack unwinding continues.

Context managers are typically used to manage opening and closing resources such as files, but they are also a good match to the work that must be done to set up a branch or a loop: At the head of the control structure, a branch condition must be evaluated, and at the end a label (and, for loops, a jump back to the loop head) must be generated. `pylibjit` defines context managers `branch` and `loop` for this purpose. Figure 3 shows how the code building the Fibonacci function can be simplified by using a decorator and the `branch` context manager. The context manager's entry function generates the labels needed to distinguish the true and false branches of execution; unfortunately, the abstraction is not perfect because the user must still take care of placing some of the jumps and labels.

This interface makes it convenient to build functions at run time and, if needed, specialize them to user input that is partly known (such as user-defined filters [Pet07]). However, the API is still too verbose for functions encapsulating algorithms that we do not want to specialize in this way. The next section puts everything together by showing how `pylibjit` can leverage Python's own syntax for specifying compiled functions.

4 Compiling Python functions: The AST approach

As the final step in the initial development, we note that Python's `inspect` module allows function decorators to access metadata for functions. In particular, it can be used to obtain the function's original source code (if the program is available in source form, which is typically the case); that code can be extracted and passed to the `ast` module to obtain an abstract syntax tree (AST) for the function [BJ13].

```
@jit.compile(return_type=jit.Type.int,
             argument_types=[jit.Type.int])
def fib(n):
    if n < 2:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

Figure 4: The compiled naïve Fibonacci function, using unmodified Python source code.

We are therefore in a position to write a decorator function which accesses its decorated function's AST and traverses that AST emitting appropriate `pylibjit` instructions for each AST node. That is, we obtain a very simple way to replace interpreted Python functions by compiled code implementing the same semantics. The Python code itself need not change at all: Whether the code is interpreted or compiled depends only on whether an appropriate decorator is present.

A compiled version of the Fibonacci function is shown in Figure 4. Note, again, that apart from the decorator it is identical to the original Python implementation in Figure 1. At any point during development, the decorator can be removed (e. g., by commenting it out) to switch back to the original interpreted function, or inserted again to obtain the benefits of compilation.

As with all the previous examples, this also produces an object that can be called like a function. On our development machine (Intel Atom N270 at 1.60 GHz running Linux 3.2.0), it takes about 8.2 seconds to evaluate `fib(32)` for the original Python version, while the compiled version takes about 0.155 seconds, for a speedup of about 53 $\times$.

5 Compiling Python functions: The bytecode approach

Compiling functions from the Python AST has both advantages and drawbacks. An advantage is that top-down AST traversal is aware of enclosing context and can generate special code for some constructs in particular contexts. As an important example, the loop

```
for i in range(n):
    ...
```

can be optimized into an unboxed counting loop with `i`'s value ranging from 0 to $\mathtt{n} - 1$. This is only possible for this `for` loop construct containing a call to `range` in the head: In all other contexts, `range(n)` must be compiled into a regular function call. This case is easy to detect in the AST. Without easy access to the information which calls occur in loop heads, our compiled code would have to follow more costly standard Python semantics: Construct an iterator from the `range` object and call its `next` method on each iteration to obtain a boxed integer object.

However, compiling from the AST duplicates some of the work that Python's built-in AST-to-bytecode compiler and its bytecode interpreter already know how to perform. In particular, when handling general boxed objects, it is difficult to generate code from the AST that exactly matches the sometimes obscure rules by which Python increments and decrements reference counts. Getting reference counting right is very important, as missing decrement operations lead to costly memory leaks, while missing increments (or too many decrements) cause memory corruption bugs.

In order to match the Python interpreter's semantics, we therefore implemented a second compiler based on its bytecode format. The bytecode representation of any Python function can be obtained via reflection and visualized using the `dis` (disassembly) module. The human-readable representation of the bytecode for the naïve Fibonacci function is shown in Figure 5. After some preliminary information about the function, the bytecode instructions are listed in blocks corresponding to source code lines (with line numbers in the leftmost column). Each instruction is listed with its offset, in bytes, from the beginning of the block of bytecode; jump targets are additionally marked with a >> symbol. The last columns show the decimal values of each instruction's numeric operand, if present. These numbers are also resolved to names of constants or variables where appropriate.

The bytecode is normally executed using a simple stack-based interpreter. The various `LOAD` instructions load operands (variables or constants) onto the value stack, while arithmetic, logic, and comparison instructions such as `BINARY_ADD` or `COMPARE_OP` consume them and push their own results.

Our bytecode compiler simulates these stack operations at compile time. In a single linear pass, the compiler visits each instruction and emits appropriate code. The implementation of the compile-time and execution-time semantics of each instruction can mostly be translated from the implementation of the instruction in the standard bytecode interpreter. For example, the Python interpreter's implementation of the `LOAD_CONST` instruction is as follows:

```
TARGET(LOAD_CONST)
    x = GETITEM(consts, oparg);
    Py_INCREF(x);
    PUSH(x);
```

This obtains the constant value `x` at a given index `oparg` in the current execution context's `consts` array, increments its reference count, and pushes it onto the interpreter stack. The following is a simplified implementation of the same operation in `pylibjit`'s bytecode compiler:

```
def LOAD_CONST(func, arg, offset):
    boxed = consts[arg]
    name = str(boxed)
    type = symbols[name].type
    x = StackEntry(boxed_value=boxed, name=name, type=type)
    INCREF(x)
    PUSH(x)
```

```
Constants:
   0: None
   1: 2
   2: 1
Names:
   0: fib
Variable names:
   0: n
  5           0 LOAD_FAST                0 (n)
              3 LOAD_CONST               1 (2)
              6 COMPARE_OP               0 (<)
              9 POP_JUMP_IF_FALSE       16

  6          12 LOAD_FAST                0 (n)
             15 RETURN_VALUE

  8     >>   16 LOAD_GLOBAL              0 (fib)
             19 LOAD_FAST                0 (n)
             22 LOAD_CONST               2 (1)
             25 BINARY_SUBTRACT
             26 CALL_FUNCTION            1
             29 LOAD_GLOBAL              0 (fib)
             32 LOAD_FAST                0 (n)
             35 LOAD_CONST               1 (2)
             38 BINARY_SUBTRACT
             39 CALL_FUNCTION            1
             42 BINARY_ADD
             43 RETURN_VALUE
             44 LOAD_CONST               0 (None)
             47 RETURN_VALUE
```

Figure 5: Python bytecode for the naïve Fibonacci function.

At compile time, the constant value is obtained from the function's constant pool and stored in a `StackEntry` object along with some useful metadata such as the constant's name and type. Code to emit the object's reference count at run time is emitted by the `INCREF` function, and the `StackEntry` is pushed onto the stack *at compile time*. In contrast to the interpreter, the evaluation stack and its entries do not exist at run time: Operands are stored in `libjit` values that are mapped to machine registers as far as possible. The compile-time stack is used to match these values with the appropriate operations.

The bytecode compiler contains a function like the one above for each instruction in the bytecode instruction set. A main loop iterates over the code and invokes the appropriate function for each instruction. While the implementation of `LOAD_CONST` is particularly simple, other instructions involve more work at compile time. In particular, arithmetic instructions make use of the metadata on the compile-time stack to determine the types of their operands, and whether those operands are boxed (stored in an object on the heap). Computations on two unboxed numbers can be compiled directly to the machine's arithmetic instructions, while other operations involve calling the appropriate Python API function. Thus the stack allows us to recover much of the context that is present in the AST, as discussed above.

However, some patterns are more difficult to detect in the bytecode. The example of the counting `for` loop iterating over a `range` of integers compiles to a sequence of several bytecode instructions:

```
          3 LOAD_GLOBAL              0 (range)
          6 LOAD_FAST                0 (n)
          9 CALL_FUNCTION            1
         12 GET_ITER
    >>   13 FOR_ITER                 6 (to 22)
         16 STORE_FAST               1 (i)
```

A scheme based purely on compiling each instruction in isolation would not be able to unbox this loop. We solve this case using a combination of laziness and lookahead: Calls to the built-in `range` function are not compiled immediately but propagated on the compile-time stack until the context in which the return will be used is clearer. When the compiler encounters an instance of `FOR_ITER` that takes such a `range` operand and is followed by a `STORE_FAST` instruction to an unboxed number, the optimized counting loop is generated without ever executing a call to `range`. Otherwise, the compiler emits the call and follows Python's usual iteration protocol.

Another obvious optimization is the elision of reference counting operations [Bru11]. The stack entry metadata can track an abstract reference count field at compile time and only insert increment/decrement operations that are actually needed because objects are stored in or retrieved from dynamic data structures. Other operations are incidental; the interpreter only needs them to ensure that all references from the stack are counted correctly. If the stack is compiled away, these operations could disappear as well. A prototype of this optimization exists in `pylibjit`, but it is not completely bug free yet and was not enabled during the experiments reported below.

Benchmark	Execution time (s) Interpreted	Compiled	Ratio
`fannkuch`	7.57	4.78	1.58
`float`	2.09	1.92	1.09
`meteor`	1.27	0.99	1.28
`nbody`	2.01	0.81	2.48
`richards`	1.02	1.01	1.01
`spectral_norm`	2.61	0.29	9.00
`AES`	12.40	0.61	20.32
geometric mean			2.69

Table 1: Influence of compilation on execution times.

The bytecode compiler is useful in supporting language features that involve complex control flow, such as list comprehensions. The AST-based `pylibjit` compiler prototype does not try to unravel these, but their bytecode representation is handled automatically using regular control-flow instructions.

As a consequence, the bytecode-based `pylibjit` compiler is able to handle a much larger subset of Python than the AST version. In principle, the only features that are not supported are default function arguments and constructs that would require us to support an explicit call stack at run time: Python's exceptions and coroutines (generators). However, we do support the special case of Python's iteration protocol which uses exceptions internally.

Overall, the implementation of the AST-based and bytecode-based `pylibjit` compilers and various shared utilities comprises about 2400 lines of Python code.

6 Experimental evaluation

We have applied `pylibjit`'s bytecode-based compiler on a number of benchmarks from Python's benchmark suite[3] and an implementation of the AES encryption algorithm[4]. Since our compiler requires type annotations for function signatures and all variables used by each function, using `pylibjit` involves some manual work and is mainly useful for programs that have a few very hot functions. We focused on such benchmarks and had to exclude others that would have required annotation of too many functions. Some other benchmarks could not be compiled because they used unsupported language features or hit bugs in our prototype compiler.

Of the benchmarks we were able to handle with `pylibjit`, we achieved speedups versus the Python 3 interpreter in all cases, although the effect is quite small in some cases. The `richards` benchmark performs a virtual method call in a very hot loop. The call

[3] http://hg.python.org/benchmarks/

[4] Adapted to Python 3 from a version available from https://bitbucket.org/pypy/benchmarks/src

is monomorphic, and while our type annotations allow us to devirtualize such calls in general, this specific case uses inheritance in a way that the compiler cannot specialize yet. We therefore have to generate a call from our compiled code into the Python interpreter; such calls require construction of an explicit call stack frame object and are therefore as slow as normal function calls in the interpreter. This kind of call also limits speedups in several other benchmarks. Other runtime features affect other benchmarks: The `float` benchmark is slightly sped up because some floating-point computations can be unboxed, but execution time is dominated by a very large number of attribute lookups in Python objects, which are performed using the standard code in the interpreter.

The programs that involve most unboxable arithmetic operations are `spectral_norm` and `AES`, which result in correspondingly large speedups of $9\times$ and $20\times$, respectively. (We have previously reported a speedup of $53\times$ on a slightly different version of the `spectral_norm` benchmark than the one used here [Bar13]. The difference is the use of `range` versus `enumerate` in a `for` loop head; unboxing is implemented for the former but not the latter.)

7 Conclusions and future work

We presented `pylibjit`, a Python wrapper for the GNU `libjit` just-in-time compiler library. Besides just exposing the underlying API, `pylibjit` allows the use of decorators to cause existing Python functions to be compiled to machine code. This is due to Python's flexibility and its large standard library that includes various useful reflective tools. Our `pylibjit` library supports a large fragment of Python that includes everything except default function arguments, exceptions, and coroutines.

The current version of `pylibjit` is not fit for general use, but the compiler is simple and extensible and is undergoing thorough debugging for a public release. Since the compiled code runs within the Python interpreter, all of the functions used internally by Python to implement operations are accessible and can be called from the compiled code. This allows code compiled with `pylibjit` to interact with any Python objects, including ones implemented in other languages in extension modules.

The presence of type annotations means that we can elide certain operations that add overhead, such as dynamic typechecks and boxing/unboxing operations. An obvious application of this work would be studies of the speedups that could be achieved using static type inference for Python. Another goal is to study the applicability and impact of optimizations such as reference count elision.

References

[asm] asm.js: an extraordinarily optimizable, low-level subset of JavaScript. http://asmjs.org/.

[Bar13] Gergö Barany. pylibjit: A JIT Compiler Library for Python. Preprint presented at Kol-

loquium Programmiersprachen (KPS), 2013.

[BCFR09] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. Tracing the meta-level: PyPy's tracing JIT compiler. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, ICOOOLPS '09, pages 18–25, New York, NY, USA, 2009. ACM.

[BJ13] David Beazley and Brian K. Jones. *Python Cookbook, 3rd Edition*, chapter Parsing and Analyzing Python Source. O'Reilly, 2013.

[Bru10] Stefan Brunthaler. Efficient interpretation using quickening. In *Proceedings of the 6th symposium on Dynamic languages*, DLS '10, pages 1–14, New York, NY, USA, 2010. ACM.

[Bru11] Stefan Brunthaler. *Purely Interpretative Optimizations*. PhD thesis, Vienna University of Technology, 2011.

[Cor11] Jonathan Corbet. A JIT for packet filters. https://lwn.net/Articles/437981/, 2011.

[CSR+09] Mason Chang, Edwin Smith, Rick Reitmaier, Michael Bebenita, Andreas Gal, Christian Wimmer, Brendan Eich, and Michael Franz. Tracing for web 3.0: trace compilation for the next generation web applications. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '09, pages 71–80, New York, NY, USA, 2009. ACM.

[num] numba: NumPy aware dynamic Python compiler using LLVM. https://github.com/numba/numba.

[Pet07] Charles Petzold. On-The-Fly Code Generation for Image Processing. In Andy Oram and Greg Wilson, editors, *Beautiful Code*. O'Reilly, 2007.

[Sel09] Dag Sverre Seljebotn. Fast numerical computations with Cython. In *Proceedings of the 8th Python in Science conference (SciPy 2009)*, 2009.

[she] shedskin: An experimental (restricted-Python)-to-C++ compiler. https://code.google.com/p/shedskin/.