Decompilation at Runtime and the Design of a Decompiler for Dynamic Interpreted Languages

Rocky Bernstein
rb@dustyfeet.com

Abstract

Decompiling is often used in conjunction with recovering lost source code, or in reverse-engineering code when we do not have access to the source.

Here we describe a novel use: places where accurate position reporting, even in the presence of optimized code or where source code is not available, could be helpful. Examples include tracebacks, debuggers, core dumps and so on. Also new is using decompilation to assist debugging at runtime.

We show how more conventional methods of source-code reporting are vague and ambiguous. Although effecting a pervasive change in a compiler is arduous and error-prone, a decompiler can be developed somewhat independently.

However, for the vast number of programming languages, decompilers do not exist. This is true even for dynamic interpreted languages where there is little going on in the way of "compilation." Traditionally, decompilers are labor intensive and ad hoc, and their construction might also be unfamiliar. So this paper shows how to ameliorate these problems by describing a pipeline for writing decompilers for dynamic languages that we hope will become standard in the same way that the pipeline from the Dragon Book [2] has.

Our pipeline differs somewhat from the standard compiler pipeline and from earlier decompiler pipelines [13, 16]. The differences may lead to further avenues of research and practice.

We use a grammar-directed parsing of instructions with an ambiguous grammar to create a Syntax Tree that, at the top-levels, resembles that of the source-code AST. This is helpful in error reporting.

An ambiguous grammar can give an exponential number of derivations on certain likely input sequences, and we describe techniques to deal with this issue.

Finally, we describe issues around developing and maintaining a Python bytecode decompiler, which handles the Python language spanning 15 releases over some 20 years.

We hope to encourage the incorporation of decompilation into tracebacks, debuggers, core dumps, and so on.

1 Problems with Existing Methods for Location Reporting, and a Solution

Reporting a line number as a position in a program can be vague. Consider the following Python code:

```
x = prev[prev[p]]
...
```

If we get the following Python error at runtime:

```
IndexError: list index out of range
```

a line number, and possibly method name or path, will be reported. But which index caused the problem?

Or consider possible runtime errors in these lines:

```
x = a / b / c # which divide?

 # which index?
[x[0] for i in d[j] if got[i] == e[i]]

return fib(x) + fib(y) # which fib?

# code created at runtime
exec(some_code)
```

As seen in the last example, there are problems with functions like *eval()* which evaluate either a string containing a program-language fragment, or which evaluate some intermediate representation of a program such as an Abstract Syntax Tree or a code object. In these cases, the program is created at runtime, either from strings or from method calls which create AST objects and, either way, there is no file containing source code as a stream of characters.

The *Pos* type in Go's AST is more informative in that it encodes the line number, file name and column number all very compactly as an opaque index [4]. Even better, though, would be to extend this to an interval or a beginning and ending position. Go's implementation cleverly takes advantage of a sparseness property: the number of interesting locations in a program, whether a single point or a pair of points, is much smaller than the program's total number of bytes. Therefore it is practical for single integer to index into a small table whose entries have a lot of information.

Long experience has shown that trying to get compilers and interpreters to add more precise location correspondences such as interval ranges, is an uphill battle. The Go implementers resist extending a *Pos* type to cover interval positions because of their fear that the line/column approach already slows down compilation.

Just adding a column position to a compiler that lacks this is, in truth, is disruptive and likely to introduce errors. The LLVM project, a fork of the GNU Compiler Collection (GCC), added a column position on *parser* errors. However, it took years before a corresponding change was added to GCC as well. Column position tracked at runtime is just as useful as at parse time.

Even when both a line and column number are indicated, there is still ambiguity. Consider:

```
fn().a
^
```

Is the problem in accessing `fn` before the call or accessing the result after?

We have observed several fruitless discussions with compiler writers wavering over which single line number to use for a particular instruction when there are several natural choices. Furthermore, different languages arbitrarily decide differently. In Python, an instruction that loads a string which spans many lines in the source code, such as a triple-quoted string, is associated with the location of the source-code line where the string ends. In Ruby, however, the location recorded is the line where the string starts. This arbitrariness could be avoided by simply giving the interval that the string spans.

For the examples given at the beginning of this section, the *trepan3k*[6] gives the following information, among other things described below:

```
x = prev[prev[0]]
        --------
x = prev[prev[0]]
    -------------
[x[0] for i in d[j] if got[i] == e[i]]
 ----
[x[0] for i in d[j] if got[i] == e[i]]
               ----
```

```
[x[0] for i in d[j] if got[i] == e[i]]
                       ------
[x[0] for i in d[j] if got[i] == e[i]]
                                 ----
x = a / b / c
          -
x = a / b / c
      -
```

This kind of precision in location reporting is common in tree-based interpreters. It is also found in some educational or teaching interpreters and IDEs with debuggers. Lisp interpreters evaluating lambda-expressions often have this kind of precision, or can without much difficulty.

What is new here is *reconstituting* a tree from instructions at runtime so we can have this information at hand. In other words, this is like a JIT for debugging position information.

There still is the fundamental and insurmountable problem of trying to associate a machine instruction with a source-code position, because there is a many-to-many mapping between instructions and source-code constructs. We have a similar problem in translating between human languages. If we translate "templado as "not hot and not cold," which part of "templado" corresponds to "not hot" and which part to "not cold"?

Although this problem is inherent, in practice we can do much better via decompilation. In addition to the spanning ranges shown above, we also give a grammar-based construct, like `array_subscript` and its context.

The instruction itself can also be helpful. In the example above with `fn()`, if the instruction is loading the method name `fn` or is at an instruction immediately before the `CALL` instruction, then we are before the call.[1] If instead the `CALL` instruction is found in a backtrace, then we are in the middle of running the call. Finally, if we are at an instruction immediately after the `CALL` instruction, then we are probably in the process of using the return value of the call.

All of this is done without modifying the compilation process. We do have to hook into the runtime environment, but only at the point of an exception, which is usually easier than changing the runtime or adding additional information. Often the language has in place a callback-hook mechanism for handling runtime exceptions.

Since this is done as a separate, isolated section of code, or perhaps an external module, we can give a very precise location without slowing down compilation or bloating the runtime system. Even more, the position is not just a range of text in the source code, but it can also be associated with grammar constructs.

In sum, the solution given here handles the problems enumerated above:

- ambiguity of location,
- source does not exist as text, and
- invasive changes to compiler and runtime environment

We close this section with another example, using the Python debugger *uncompyle6* [6] which decompiles at runtime. For this simple-minded Fibonacci function:

```
def fib(a):
    if a < 2: return 0
    return fib(a-1) + fib(a-2)
```

here is an example of the interaction with the debugger, showing positions:

[1] See [20] for a description of Python bytecodes; however for formatting reasons we shorten the `CALL_FUNCTION` instruction name to `CALL`.

```
(/tmp/fib.py:1): fib
-> 1 def fib(a):
(pdb) up
(/tmp/fib.py:3 @30): fib
(pdb) backtrace
##0 fib(a=0) called from file 'fib.py':1
->1 fib(a=2) called from file 'fib.py':3
##2 <module> exec()
     '/tmp/fib.py' at line 5
(pdb) deparse -p
instruction: 30 CALL_FUNCTION_1  1
grammar LHS: call_function
return fib(a - 1) + fib(a - 2)
                    ----------
Contained in...
Grammar Symbol: binary_expr
return fib(a - 1) + fib(a - 2)
       ----------------------
```

We went an entry up in the call stack. The output of running the `deparse` command gives us:

- The instruction where we are stopped. In the example above, it is the `CALL_FUNCTION` instruction;
- The source-code text associated with this; and
- The larger context which is a `binary_expr`, here an addition.

Debuggers for Python 2 [11], Python 3 [6] and Perl [8][9] all have commands which make use of this technology.

2 A Deparser Pipeline for Dynamic Languages

Most programming languages lack a decompiler, or at least a good one, although there seems to be a constant need for them. Besides the novel use described in the last section, decompilers are popular in the reverse engineering and vulnerability testing communities for obtaining source code when none is available.

In the technology of programming languages, a number of things have changed in the last couple of decades which make writing decompilers easier:

- Many dynamic languages interpret rather than run binary code. Some of these are somewhat (but not totally) general-purpose interpreters, such as JVM[23], which work at a higher level than machine code.
- Others of these have tree evaluators so the structure is present at runtime (examples: Perl[18], Ruby 1.8 and Korn shell.)
- Finally, others have bytecode interpreters with instructions which are custom to the language that a decompiler can make use of (examples: Python[20], Lua[27, 28], Emacs Lisp)[2]
- There is very little in the way of translation or "compiling" from the source code to byte code or tree representation that is used at runtime. Often the code generation is done in a single pass in a single program file, which limits the complexity of the translation.
- Improvements have been made to general context-free parsers which allow them to run fast (in linear time with a low constant overhead).

In this section we describe the plan of a pipeline for a decompiler which we feel will fit the many interpreted dynamic languages such as those listed above.

Figure 1 compares a proposed decompiler pipeline used in the *uncompyle6* [12] and Emacs Lisp decompilers, to the conventional Dragon Book pipeline. The dashed lines show correspondences between phases of the two pipelines. Dotted boxes show phases that are optional.

There are some small but important differences between a compiler and an instruction-based (or non-tree based) decompiler, namely:

- We do not start out with program text but executable instructions of some sort.
- We can assume that what we are given is correct, which means ...
- We do not have to understand everything completely. For example we can ignore bookkeeping instructions (popping a stack value to make it match a join with another branch).

[2]For many of these languages, there are several different VM interpreters.

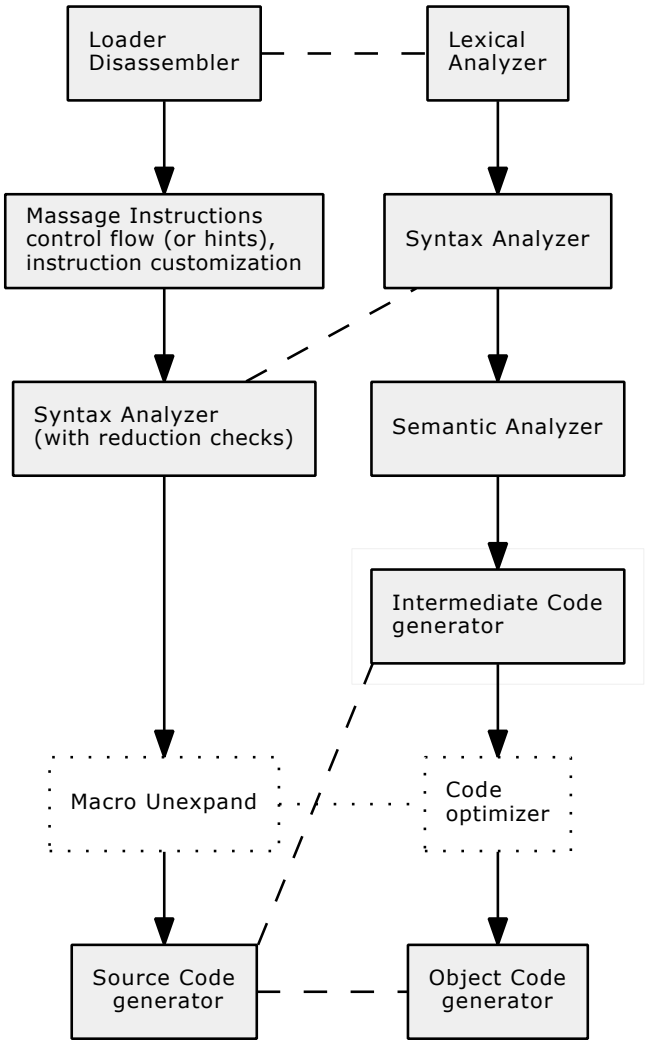

Figure 1. Decompiler vs. Dragon-Book Pipeline

- We do not have to perform type checking

In what follows, we discuss these specific phases and their difference from "conventional" pipelines:

- program loading and instruction disassembly
- token massaging, e.g., to speed up parsing and to detect control flow
- parsing using an ambiguous grammar. Custom grammar rules may added; further checks at potential reduction to disambiguate
- macro un-expansion when the language has macros or, more generally, high-level source code rewriting.
- recreating source text; mapping instruction offsets to fragments of text when exact position information is desired

2.1 Loader/Disassembly Phase

As shown in figure 1 we correlate this phase with Lexical Analysis in a conventional compiler pipeline.

In conventional compiler technology we scan characters and build up tokens, like identifiers or strings. Tokens have an additional attribute like 5 for a number token or "mango" for a string token.

In our decompiler, tokens are decoded or symbolic forms of assembly instructions. Getting to the stage where there is a sequence of instructions is straightforward but can require a number of steps. Either object code is read from some bytecode or object file and "unmarshaled" into Python objects, or code is already loaded into memory. In any case, after that a stream of bytes generally has to be decoded by unpacking the stream into instructions which are then further broken up by looking

up opcodes, computing jump addresses, and extracting symbolic names for operands. The whole process is similar in spirit to regular expression parsing in that, however involved, the overall time complexity is approximately linear.

At the end of this process, there is a stream of instructions, grouped by modules, classes, files and/or functions, possibly in some hierarchical way. For each instruction there is its opcode name, its operands and possibly other information. The resulting stream of instructions corresponds to a stream of tokens in a conventional parser. In some rare cases this can be fed directly into a parser. More often, it needs some additional massaging.

2.2 Massage-Instruction-Stream Phase

As shown in figure 1 this phase does not have a counterpart in a conventional compiler pipeline.

Except in rare cases, decompilers cannot simply take the output of a disassembly and feed it to a parser. Instead they must modify the instructions to make ambiguous parsing possible and more efficient, for two broad reasons:

- to reduce the number of parses
- to make it possible to detect structured control flow via parsing

2.2.1 Grammar Ambiguity

Decompilation resembles human-language translation in three ways. There are generally many valid translations. Furthermore, judgment of their quality is subjective (except when a retranslation is exact, which is often the case in certain dynamic languages). Finally, a translation might not need to be exact to be useful.

In many programming languages, nested `if`'s are semantically equivalent to a single `if` statement with short-circuit `and` expression. That is:

```
if (condition1) {
  if (condition2) {
      ....
  };
}
```

is the same as:

```
if (condition1 && condition2)
   ...
  };
}
```

A decompiler can sometimes determine which form was used in the source code by using idiosyncrasies of the compiler. In many interpreters, since there is very little in the way of compilation and analysis, there are two constructs and the template for these will be different in order to cover the most generic situations. Even though the templates are different, after instantiation the semantics are often the same.

In CPython for example, the `if` blocks may have additional scope-setup instructions, whereas a short-circuit `and` expression does not use these instructions. Also, CPython sometimes leaves dead code around as the remnants of a template which had empty parts. *uncompyle6* takes advantage of such situations and matches the templated dead code when it arises.

But in general, a decompiler is better served by coping with the multiple equivalent translations. This suggests that the grammar which transforms instructions into source code is ambiguous.

Traditionally, general context-free grammars have been discouraged or have gotten little coverage in compiler books. A number of compiler books [1, 2, 32] mention the Earley parser algorithm only in end notes.

In the last decade or so, there has been a rise of Parsing Expression Grammars (PEG)[17] and it has been shown that parsers for context-free grammars can be made efficient [24, 31].

There is however a new kind of problem which arises when working with ambiguous grammars: limiting the number of valid parses.

This problem generally does not arise in conventional compilers because it is desirable that unique source texts have unique derivations. Therefore, compiler writers produce grammars which are designed to have that property, i.e., grammars which are unambiguous. In contrast, in our Alice-Through-the-Looking-Glass world we start from a set of instructions which are presumed to be valid, and the decompiler writer needs to come up with a grammar which covers that. It is even fine to have a grammar that accepts sequences of instructions which can never exist in practice. In sum, ambiguous grammars make it easier to get coverage.

But when we say, for example, that an Earley parser has time $O(n^3)$, we mean that the time for a *single* derivation is bounded by $O(n^3)$. If there is an exponential number of derivations, then the time could conceivably be super-exponential.

Certain kinds of simple grammars can have an exponential number of derivations. To see this, consider the following simple context-free (and non LR- or LL-) grammar:

```
E ::= E E | a
```

On a string of n a's for $n > 1$, there are $2^n - 1$ ways to derive that string. At each step we can decide to expand the first or the second E. So, for number of derivations $T(n)$ we have:

$$T(1) = 1 \tag{1}$$

$$T(n+1) = 2T(n) \tag{2}$$

This recurrence relation is $O(2^n)$ which means we will need that order of time and space.

This would be an idle curiosity except that in stack-based instruction architectures, such situations can arise. In particular, instructions with variable-length arguments, such as call functions, have this behavior. The following grammar for handling Python's variable-length CALL instruction creates a non-linear set of grammar derivations:[3]

```
expr ::= c_expr | LOAD
c_expr ::= CALL | expr CALL | expr expr CALL | ...
```

If we encounter an instruction sequence like:

```
LOAD LOAD LOAD LOAD CALL CALL
```

we will run into such a combinatorial explosion.

The remedy to turn this back into a linear parse is simple: we add custom grammar rules. In CPython, the operand of a CALL instruction is the number of arguments in the function. By adding only the specific forms that we see in a particular function, we greatly reduce the number of derivations tried and the number of grammar rules.[4] For example if the first CALL's operand said it took one argument while the second CALL's operand said it took four arguments, we would add a grammar rule at runtime in the instruction-massage phase:

```
c_expr ::= expr CALL1
c_expr ::= expr expr expr expr CALL4
```

Also we would modify the input instruction sequence to:

```
LOAD LOAD LOAD LOAD CALL1 CALL4
```

With this, there is just one way the sequence can be derived:

```
LOAD LOAD LOAD LOAD CALL1 CALL4 ->
LOAD LOAD LOAD expr CALL1 CALL4 ->
LOAD LOAD LOAD c_expr CALL4 ->
LOAD LOAD LOAD expr CALL4 ->
...
expr expr expr expr CALL4 -> c_expr -> expr
```

2.2.2 Control Flow

In decompilers for compiled languages, recreating the high-level control structure has to be done from first principles: creating basic blocks by looking at instruction jumps and at instructions where jumps meet and, with this and the control-flow graph in hand, applying an additional algorithm. For the class of interpreters we consider, this job can largely be done by grammar pattern matching, and in these cases, the end result will match the source code more exactly. Also, it provides us with an AST that can be used in location reporting; see section 1.

When there is little or no branch optimization, e.g., changing the targets of an unconditional jump instruction that jumps to other unconditional jump instructions, we can often come up with a grammar that does not need special treatment of control flow. The nesting and sequencing of compound statements follow well what grammars can track.

Early versions of Python did not have branch optimization, so their decompilers did not need much in the way of control-flow analysis. Over time the compiler for each CPython release performed more branch optimization. Any serious decompiler for Python 3.6 or later will need some sort of control-flow analysis.

uncompyle6 keeps the grammar approach, but adds bracketing pseudo-instructions where needed. The first one introduced was COME_FROM. Later the name was refined to indicate the larger compound structure, e.g., COME_FROM_LOOP

[3] The Python opcode names are really CALL_FUNCTION and LOAD_CONST but again for formatting purposes, we shorten them to CALL and LOAD.

[4] Without customization, there could be an exponential number of grammar rules.

Inspection of a bytecode's offset is sometimes all that is needed to distinguish two similar kinds of Python constructs. For example, a Python `break` statement is distinguished from a `continue` statement by considering the target of the corresponding jump instruction. So, control-flow analysis decides which of these situations is relevant and changes a `JUMP_ABSOLUTE` instruction into a pseudo-instruction, `BREAK` or `CONTINUE`.

2.3 Syntax-Analyzer Phase

As shown in the pipeline of figure 1 this phase corresponds to the Syntax Analyzer phase in a conventional compiler pipeline. The major difference is that we must use an ambiguous grammar.

The grammar used to describe a programming language is usually unambiguous because we do not want there to be several possible meanings for a given program. However, ambiguous grammars are appropriate for a decompiler reconstructing the source code. In section 2.2.1 we discussed why it is useful to have a full context-free grammar, along with the issues around ambiguity and the time and space complexity which it can incur if not handled properly.

Ambiguity in grammars can also be used simplify the grammar. It might introduce erroneous meaning, as opposed to multiple valid meanings, but the simplification can be helpful. Handling ambiguity via operator precedence is an often-cited use case [2].

Here we give another situation where ambiguity simplifies things, and use that to show how we choose among several grammatically valid parses. Python offers a lot of flexibility in calling a function. It allows positional arguments and keyword arguments.

Part of our (ambiguous) grammar rule for handling a call is:

```
kwarg   ::= LOAD_CONST LOAD_CONST
pos_arg ::= LOAD_CONST
c_expr  ::= pos_arg pos_arg CALL | kwarg CALL
```

For input:

```
LOAD_CONST LOAD_CONST CALL
```

By looking only at the opcode of instructions, we cannot tell whether this is a call with two positional arguments or a call with one keyword argument. We also have to look at a flag in the operand of the `CALL` instruction.

We have extended the Earley algorithm parser so that it can perform checks at parser-reduction time. The technique we use is new and is not described in the literature. We can register the callback for a rule such as:

```
c_expr ::= kwarg CALL
```

In registering, we can specify that either the callback should be passed the sequence of tokens forming the rule, or that an AST built from the sequence of tokens should be passed. The callback can then run any code to check for validity. The parser performs the grammar-rule reduction only if the callback routine deems the reduction valid.

The details of what goes on in *uncompyle6* are a bit involved, but this is the essential idea. Reduction-rule checks in *uncompyle6* also assist in disambiguating control flow.

2.3.1 Making Grammars look Similar

This may be obvious, but it is worth mentioning. At the upper levels, the nonterminal names for the AST used by both the compiler and the decompiler should be the same where possible. For example, a programming language generally has statements, and those can be simple, e.g., *assignment* or *call* statements, or compound statements, such as *if* or *loop* statements, and so on.

We described in section 1 that as part of the error location we can give a nonterminal name for context. Many languages, such as Python [19] and Go [3], have AST modules which assign standard names for various constructs.

Unavoidably, though, there has to be some point at which these grammar rules differ since the end results, the source language and the object-code language, are different.

In handling multiple grammars for what is logically a single language with slight variations between versions, we can add production rules to keep the upper levels of the AST looking the same.

As a specific example, the Python 2.6 sequence

```
JUMP_IF_TRUE POP_TOP
```

was combined into a single instruction with the opcode

```
POP_JUMP_IF_TRUE
```

in Python 2.7. So rather than have this rule for the Python 2.6 grammar:

```
testtrue_then ::= expr JUMP_IF_TRUE POP_TOP
```

and this one for the Python 2.7 grammar:

```
testtrue_then ::= expr POP_JUMP_IF_TRUE
```

we have a single rule for Python 2.6, 2.7, and others:

```
testtrue_then ::= expr jmp_true
```

with the corresponding rules for `jmp_true`, e.g., for Python 2.6:

```
jmp_true ::= JUMP_TRUE POP_TOP
```

and for Python 2.7 the singleton rule:

```
jmp_true ::= POP_JUMP_IF_TRUE
```

Although this adds levels to the generated tree, it simplifies semantic actions. For example, in the Python grammars `jmp_true` is also used in this rule:

```
assert ::= assert_expr jmp_true
    LOAD_ASSERT RAISE_VARARGS_1 come_froms_pop
```

By using a single nonterminal here rather than one or two terminal symbols depending on grammar version, the ordinal position of `LOAD_ASSERT` is kept the same. The semantic actions need to pull out the attribute of that instruction, and having a fixed location of it helps. So, just one set of semantic-action routines largely covers all grammars.

2.3.2 AST-building Notes

AST building pretty much matches the corresponding action in a conventional compiler. Below we give some notes of things we have encountered. Some of these may be obvious to someone familiar with the corresponding action in a conventional compiler.

To reduce the depth of the tree, we remove singleton derivations between nonterminals. The semantic actions are aware of this. We handle operator precedence using a table of priorities.

Another AST simplification is to mark certain nonterminals as being list-oriented. When the AST tree builder is building nodes for these, it appends them to a list; for example:

```
stmts -> [stmt, stmt, stmt]
```

rather than creating a chain such as:

```
stmts -> [stmts, stmt]
            |
            +--> [stmts, stmt]
                   |
                   +--> stmt
```

There is another, practical side to this. Since the AST is traversed recursively, the list can be traversed more efficiently by iterating over a list, and we save on stack space. Without this, the decompiler would die on a program with thousands of statements.

2.4 Macro Unexpand Phase

Some languages like C or Lisp allow macro expansion. Some perform macro-like expansion via generic-function inlining. In our pipeline, at this stage we have an AST whose upper levels resemble the corresponding AST from the source code after any macro-like expansion. Undoing macro expansion is easier at this higher level than at the level of VM instructions.

Other high-level source-to-source types of translations can also be done here. Some examples are un-inlining code or deciding which of two equivalent high-level constructs can be used.

In languages that use some kind of macro expansion, information relating what is expanded and where is usually not saved. But as a general rule, macros expand rather than contract, and the macro expansions that a programmer often most cares about are the ones that expand a great deal. Therefore any process that shortens the result will probably be beneficial, whether or not it was initially used.

For limited macro-expansion systems like the C preprocessor [33], inverting the search and replace is, in theory, reversible: we can swap the search pattern with the replacement pattern. Furthermore, the reversing process can be generated automatically; that is, a system can be devised to look for a set of macros that are either found in the program or are typically used, and from this it generates the reversing macros and applies the reversing macros iteratively. If we apply a reversing macro

only where the result yields something shorter, the processes will terminate after a finite number of iterations. Otherwise we can put a limit on the number of applications.

In more general macro-expansion systems, such as those for Common Lisp, undoing expansion may be impossible. But even here, there are often many cases where a simple search and replace would be helpful.

Further research on specific effective techniques and transformation systems for this phase may be in order. However there is a body of work and literature on program transformation systems.

In figure 1, this phase corresponds to Code Optimization in a conventional pipeline. The difference is that we are working from an AST rather than from some intermediate representation. Simpler compiler pipelines, where there is no IR or separete code-optimization phase, combine the last three boxes so that they resemble the decompiler's Source Code generator phase.

2.5 Source-Code Generator Phase

This is the last phase in our pipeline and although we list it as equivalent to the Object Code generator phase, is possibly more similar to the Intermediate Code generator in that there is a syntax-directed translation from the AST.

As mentioned in section 2, we do not have to use all the pieces of the AST, but only enough to recreate the source. (In this sense the tree created from instructions is not all that abstract: in contrast to a conventional AST for a high-level language, few if any of the input tokens or instructions have been discarded.)

In any compiler that maps high-level constructs (i.e., more abstract, and thus shorter, sequences) onto longer, lower-level constructs, we expect that there will be places where a token value might appear several times within the lower-level instruction sequence. To reconstruct source code, the decompiler needs to find *any* one of those places in the instruction sequence and can assume without checking that all other places will be consistent with that.

Here is an example. Suppose we have a loop that iterates over variable *i*. There may be code to initialize *i* before entering the loop and another instruction to increment the variable *i*. But in the source code the variable *i* appears only once:

```
for i in range(10):
   print(i)
```

This particular situation does not occur in Python, but duplication occurs in other complicated ways. Specifically, function information can be obtained from either part of Python's code structure for that function or from the instructions that dynamically create the function.

3 Debugging Optimized Code

There are difficulties in debugging code that has been optimized. This is a problem which has long been recognized, and there is no universally accepted approach. While what we describe here is not a full solution, in practice we find it useful.

The difficulties arise because a really good optimizer will scramble the program so that a one-to-one mapping between instructions and source-code constructs occurs less often—source-code instructions might be moved around, duplicated, or eliminated. On the other hand, an instruction might correspond to several different places in the source code all at once, or an instruction might represent an aspect of execution that is implicit in the program and therefore has no corresponding text. Stack clean-up on leaving a block of code is an example of this. (We cannot associate this with the end terminator for the compound statement, for example, because Python does not have such a text construct, and a single newline can close several compound statements.)

Debuggers can be complex, so it is understandable that most debuggers do not delve into the optimization steps.

Here is a canonical tricky problem in debugging optimized code. Suppose the source code looks like this:

```
if condition then
   # some unconditional code
   a :=  x / z
else
   # some more unconditional code
   b :=  (x / z) + 2
end
```

and suppose an optimizing compiler decides it is safe to rewrite this as:

```
t1 := x / z
if condition then
   # some unconditional code
   a := t1
```

```
else:
   # some more unconditional code
   b := t1 + 2
end
```

Suppose we get an error in the hoisted code, specifically a divide-by-zero error in evaluating `x / z`.

If we try to figure out what went wrong, we are probably trying to assess why `z` is zero. We really do not care about all the code after the `t1` assignment in the hoisted code. In fact, it is helpful to know that we can simply ignore those things. What we are interested in is how `t1` got its value, and that is exactly what is shown in the decompiled code.

The assignment to `t1` might look like a puzzle since we did not write `t1` anywhere. It would be nice if optimizers logged the transformations they perform in a programmer-friendly way, so we would know that this code was hoisted. Unfortunately, no optimizer in widespread use does this in a way that is accessible to a programmer at runtime when an error is encountered.

But if we had this information, a decompiler could help. It has a parse tree of the source code, so it could feed the source code or AST to some sort of code comparison analyzer which also has access to the original source.

Programmers typically perform these steps consciously or unconsciously when debugging optimized code. They do a mental decompilation to understand what the program was actually doing. We have just assisted a step by turning what might be an unfamiliar language—bytecode or machine instructions—into a familiar programming language.

In other words, one way a computer can assist debugging optimized code is to describe what is there as simply and naturally as possible. Then come the additional steps of untangling or explaining the transformations.

We believe that if we present the code simply, the way the computer sees it, and if we have reference to the original source code, we will be able to understand why the two are equivalent. And in some cases, we might decide to rewrite the program along the lines of the transformed program.

As an example, for the following Python code:

```
if x:
    return x
else:
    return f(x)
```

when we use *uncompyle6* to decompile the bytecode produced by the CPython interpreter, equivalent source code is produced:

```
return x or f(x)
```

We may decide we like this formulation better. On the other hand, if we had started with:

```
JUMP_ABSOLUTE = 100
...
if op == JUMP_ABSOLUTE:
```

this deparses to:

```
if op == 100:
```

Here we probably want to keep our original formulation, as it was intentionally more verbose for clarity and flexibility.

The kinds of peephole optimizations done by the Perl interpreter are often undoable and the Perl decompiler makes the reverse transformation when it can. In Python some reversable peephole optimizations can be covered by additional grammar rules.

However when an optimization simplifies code, that transformation is not reversable. An example of this in Python is where this code:

```
if 1:
  statement
```

gets simplified in bytecode to:

```
statement
```

Given the line number associated with the original source code, which is often in the bytecode, sometimes we can infer what transformation the Python interpreter made. In the above example, Python might leave around dead code or a `NOP` instruction which has the line number where the `if 1` was removed.

In contrast to this simple approach using decompilation, most of the other approaches to debugging optimized code involve changing the compilation system to include additional information. Again, this can be an adjunct to decompilation information.

Our experience is that debugging optimized code using decompilation is a big improvement over the alternatives.

4 Mechanics

4.1 Code Reduction

A big challenge in the Python decompiler code base is maintaining it as new releases come out. Right now there is a new Python release about every year, and the code currently supports about 15 releases. Often, both the programming language and bytecode change.

It is desirable to store changes as sets of differences rather than as fully independent code for a number of reasons:

- Changes in opcodes mean changes in grammar rules, and using differences makes the correspondence clearer; similarly …
- Changes in grammar can lead to changes in semantic actions.
- When a bug is found, naturally it is found a particular release. Tracking the differences facilitates understanding what other releases may be affected.
- Noting the differences between releases aids in writing other tools. For example, we found that when writing a bytecode-to-bytecode converter totally independent of the decompiler, it was helpful to have the version-to-version differences listed.

uncompyle6 handles this via Object Oriented subclassing. The base class is not the earliest version of Python. Instead we use some middle version for each of the Python 2 and Python 3 releases and work from that.

Usually changes are small, but not always. Because Python 2 is very different from Python 3, the base classes are different. There is some overlap, so overlap code is duplicated. We have a way to dump the full grammar for any given Python release. Therefore if we want to rebase which base class to start with, or want to create an additional base to hang Python releases from, we can.

Whenever we work with differences, we need a way not only to add things, but also to remove things. As a result, although a bit unusual, we have extended the Earley algorithm parser so that it can remove grammar rules.

4.2 Testing

As any compiler developer knows, testing is important. It is also generally time consuming since a lot of data or programs need to be tested and the compilation process is usually not very fast.

We use common-practice techniques for testing, some of which are often used in software engineering but need a little adaptation for compilers. Some techniques that are in common use in compiler development also require adaptation.

Since running a full test suite takes a long time, we run quick unit tests first. These target specific bits of code precisely, so when they fail we generally know what isolated section of code to look at and fix.

As a practical trick to speed up testing, we use three free (for open-source projects) continuous-integration services: Travis, CircleCI and Appveyor. These help with running massive regression tests in parallel. Travis can cover many different Python releases which are run in parallel, but only a limited number. On CircleCI we do massive testing of a single release using the full installed Python system library. Appveyor handles testing on Microsoft Windows. All of these CI services support more parellelism for a fee.

Another testing technique from software development is measuring code coverage. In *uncompyle6*, some of the "code" is grammar rules, so we have extended the parser system to record which grammar rules were used in the reduction, and that is how we test grammar coverage. As mentioned above, rules get pulled in via subclassing, and looking at grammar coverage has been helpful in deciding what additional tests are needed, which rules should go where in the class hierarchy, and which rules to remove. By comparing unused grammar rules across consecutive Python versions, and taking into account grammar rules added between them, we can glean the bigger picture of how code generation has changed between Python releases without having to look at the Python compiler source code.

A common testing technique used especially in compiler testing is *round-trip testing,* or performing some series of transformations which should lead back exactly to the starting point. Applied here, round-trip testing starts from bytecode, not source code. Thus the round trip is to decompile bytecode, compile the result and then check against the original bytecode, allowing for some fuzzing.

For early versions of Python where there is no optimization, this works well. However for later versions of Python this works increasingly poorly. But doing only two-thirds of the round trip will still find errors: decompile bytecode, compile source code. In the second step, compiling source code, we often find errors because the source code we produce is not valid.

For the limited set of programs that we know will "round trip" successfully, we test starting with bytecode.

In Python, as in other large systems, there already is a test suite for checking the distributed library code. Since those programs are self checking, those programs too can be decompiled and run.

5 History

Decompiling is not a new idea. Cifuentes' Ph.D. thesis indicates that it goes back at least 55 years. Most of the research on decompilers focuses on compiled and statically typed languages.

The Python code base spans 15 versions and goes back almost 20 years, and through that long period there were a number of anonymous maintainers. In a language like Python, there is a lot of churn from version to version and keeping the decompiler up to date is a bit of a challenge. In section 4.1 we describe how to deal with this issue.

The earliest authors and developers of the Python decompiler were well versed in compiler technology, but later maintainers of the code, less so. As a result there were hacks and patches that would have been better done within the framework of the parsing technology rather than ad hoc. Broader documentation and education was a motivation for this paper.

Like Python, a decompilation package for Perl [30] goes back almost 20 years. However, the decompiling techniques in the decompilers for these two dynamic languages are very different. Perl5's evaluator is tree based. This makes decompilation easier: no parser is needed to build an AST since that is already in place. Also, there has been less change from release to release of the instructions over the course of Perl's 30 years or so. Decompilation in Perl is more ad hoc than in Python.

6 Comparison with Other Work

The idea of using decompiling for better location reporting arose from discussions with the Perl community, but the implementation is new. We describe related work and ideas in section 1.

Decompiling technology is not new in either Perl or Python, but using it to find a more precise location at runtime, say in a debugger or in a stack trace, is. In order to do this we need to keep around associations from offset to text fragments, in addition to the other information. What was a batch process is now an online process. For Perl, see [7]. The parse tree for the functions that are contained in the call stack are kept, to give parent context for the fragment in question.

Also somewhat novel, at least for Python, is improving technology and testing to track the changes in the language and its bytecode over their long and variable history.

The overall dynamic decompilation structure given in section 2 is embodied in John Aycock's Python decompiler. This includes massaging the token stream to facilitate grammar parsing, and adding custom grammar rules for each function. However there is no discussion in the literature of why this is important, as in section 2.2.1 above. The decompiler pipeline is also used in an Emacs Lisp decompiler [10].

Section 2 compares the proposed pipeline with a conventional compiler pipeline. It is a little different from the "standard" decompiler pipelines described by Michael James Van Emmerik in his Ph.D. thesis [16]. Our focus is on dynamic, not statically-typed and interpreted, languages which somewhat simplifies and changes things.

Van Emmerik's diagram on page 62 shows a decompiler pipeline as a compiler pipeline in reverse, rather than being just a different kind of compiler with correspondences in the forward direction. The diagram on page 61 looks similar to ours especially in noting that the first step is loading and disassembly. However the next three middle IR steps are different, and are similar to other decompiler pipelines [15].

The pipeline in Cifuentes' Ph.D. thesis has labels that make it look remarkably like ours and a conventional compiler's. However her first 3 steps refer to assembly-language parsing. In our diagram we lump these under "disassembly," and conventional disassemblers do not resemble compilers in how they work. Like Van Emerik, Cifuentes works from an IR. The decompilers we have used do not use an IR, although we do classify instructions into Python-release independent categories, extending what is already provided by the Python *opcode* library [20]. (The bytecode itself is already an IR.)

Instead of an IR, there is simply an augmented or modified sequence of instructions. This is then fed into a parser for syntax-directed translation which does most of the heavy lifting. That is a key difference between the decompiler we propose and those described in the literature or available software [14, 15, 21]. The benefits we gain from using conventional compiler technology are the same benefits gained in a conventional compiler, namely that we are using domain-specific languages which clearly represent the problem.

We use an Aycock/Horspool [5] implementation of the Earley parser. It is customizable at runtime per function and has been extended to handle selection of rules caused by the ambiguity of the grammar in a way that is convenient for decompilation. Adding precedence rules to handle operator grouping ambiguity statically is well known. Other ways of handling ambiguity at runtime or based on the input sequence are non-standard and less well known.

There are a number of other fast and general context-free grammar parsers that could be used. Marpa [25] is one, and it also has a means for handling ambiguity. Another interesting parser generator is Nez [26, 29]. It has parsing conditions that can be used instead of those described here. Its AST constructions might be useful in macro unexpansion.

Debugging optimized code has a long history; there is no definitive solution [22, 34]. Approaches described in the literature are to add side information which a debugger can use, and perform work at runtime to unscramble the code. The approach

we suggest here is different and simple: be transparent about the code and translate it back into a language familiar to the programmer. This is not a complete solution to debugging optimized code. However, it does separate the boring and tedious parts that a programmer currently has to perform from the higher-level de-optimization transformations that may require a human. And in contrast to most of the proposals in the literature, it can be done without any changes to the optimizing compiler.

7 Summary

We introduce the idea of using decompilers to show the location of a program at any given time, and we suggest that decompilers can be used in runtime error analysis, stack traces, and debuggers, even when the code has been optimized.

Next we present the design of a decompiler for dynamic interpreted languages. It is much like that of a conventional compiler, possibly simpler. In contrast to the decompilers for statically-typed compiled languages, a syntax-directed translation using an ambiguous grammar is desirable for interpreted dynamic languages.

We describe why with an analogy to human translation. And this analogy suggests an avenue of research that might be fruitful: use Natural Language Processing techniques for examining a large body of (virtual) machine instructions, along with an AST derived from the corresponding source code, to come up with initial grammar rules for a deparser.

After the discussion of issues around using an ambiguous grammar, we discuss the day-to-day issues of testing.

We hope that this paper serves to popularize decompilers for dynamic languages and encourage their proliferaion. These additional decompilers can be used to reduce the ambiguity of error reporting and assist in debugging optimized code. There is still much to do.

8 Acknowlegement

I greatly appreciate the help of Stuart Frankel, Ph. mad., for transforming my great jumble of thoughts into something slightly less jumbled.

Others have also offered some suggestions anonymously.

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2007. *Compilers, Principles, Techniques, and Tools* (2nd ed.). Addison-Wesley, Reading, Mass. 5–6, 301–301 pages.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers, Principles, Techniques, and Tools* (1st ed.). Addison-Wesley, Reading, Mass. 10–10, 277–277 pages.

[3] GoLang Authors. 2017. The Go Programming Language; package ast. (2017). https://golang.org/pkg/go/ast/

[4] Golang Authors. 2017. type Pos. (2017). https://golang.org/pkg/go/token

[5] John Aycock and R. Nigel Horspool. 2002. Practical Earley Parsing. *Comput. J.* 45, No. 6, 6 (2002), 620–630.

[6] Rocky Bernstein. 2018. (2018). https://pypi.python.org/pypi/trepan3k

[7] Rocky Bernstein. 2018. B::DeparseTree. (2018). https://metacpan.org/pod/distribution/B-DeparseTree/lib/B/DeparseTree.pod

[8] Rocky Bernstein. 2018. Devel::Trepan. (2018). https://metacpan.org/pod/Devel::Trepan

[9] Rocky Bernstein. 2018. Devel::Trepan::Deparse. (2018). https://metacpan.org/pod/Devel::Trepan::Deparse

[10] Rocky Bernstein. 2018. elisp-decompiler. (2018). https://github.com/rocky/elisp-compiler

[11] Rocky Bernstein. 2018. trepan2. (2018). https://pypi.python.org/pypi/trepan2

[12] Rocky Bernstein. 2018. uncompyle6. (2018). https://pypi.python.org/pypi/uncompyle6

[13] Cristina Cifuentes. 1994. *Reverse Compilation Techniques.* Technical Report. Queensland Institute of Technology. 8–8, 35–36 pages.

[14] Arnaud Delobelle. 2011. (2011). https://code.google.com/archive/p/unpyc3/

[15] Robin Eklind. 2017. (2017). https://github.com/decomp/decomp/

[16] Michael James Van Emmerik. 2007. *Static Single Assignment for Decompilation.* Ph.D. Dissertation. The University of Queensland. https://yurichev.com/mirrors/vanEmmerik_ssa.pdf

[17] Bryan Ford. 2004. Parsing expression grammars: a recognition-based syntactic foundation. In *ACM SIGPLAN Notices*, Vol. 39. ACM, 111–122.

[18] Perl Foundation. 2017. PerlGuts Code Tree. (2017). https://perldoc.perl.org/perlguts.html#Code-tree

[19] Python Software Foundation. 2017. ast âĂŤ Abstract Syntax Trees, Abstract Grammar. (2017). https://docs.python.org/3/library/ast.html#abstract-grammar

[20] Python Software Foundation. 2017. Python Bytecode Instructions and Opcode collections. (2017). https://docs.python.org/3.6/library/dis.html

[21] Michael Hansen and Darryl Pogue. 2017. pycdc: C++ python bytecode disassembler and decompiler. (2017). https://github.com/zrax/pycdc

[22] Urs Hölzle, Craig Chambers, and David Ungar. 1992. Debugging Optimized Code with Dynamic Deoptimization. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation (PLDI '92).* ACM, New York, NY, USA, 32–43. https://doi.org/10.1145/143095.143114

[23] Oracle Inc. 2017. The Java Virtual Machine Instruction Set. (2017). https://docs.oracle.com/javase/specs/jvms/se9/html/jvms-6.html

[24] Jeffey Kegler. 2011. Marpa, A Practical General Parser: The Recognizer. (2011). http://dinhe.net/~aredridel/.notmine/PDFs/Parsing/KEGLER

[25] Jeffey Kegler. 2011. Marpa-R2. (2011). https://metacpan.org/pod/distribution/Marpa-R2/lib/Marpa/R2.pm Available from CPAN under the name `Marpa::R2`.

[26] Kimio Kuramitsu. 2017. nez. (2017). https://github.com/nez-peg/nez

[27] Dibyendu Majumdar. 2015. Lua 5.3 Bytecode Reference. (2015). https://github.com/dibyendumajumdar/ravi/blob/master/readthedocs/lua_bytecode_reference.rst
[28] Kein-Hong Man. 2006. (2006). http://luaforge.net/docman/83/98/ANoFrillsIntroToLua51VMInstructions.pdf
[29] Tetsuro Matsumura and Kimio Kuramitsu. 2016. A declarative extension of parsing expression grammars for recognizing most programming languages. *Journal of Information Processing* 24, 2 (2016), 256–264.
[30] Stephen McCamant and Malcolm Beattie. 2017. B::Deparse. (Sept. 2017). https://metacpan.org/pod/distribution/B-Deparse/lib/B/Deparse.pod Available from CPAN under the name `B::Deparse`.
[31] Kota Mizushima, Atusi Maeda, and Yoshinori Yamaguchi. 2010. Packrat parsers can handle practical grammars in mostly constant space. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. ACM, 29–36.
[32] Linda Torczon and Keith Cooper. 2008. *Engineering A Compiler* (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. 149–149 pages.
[33] Wikimedia. 2017. C preprocessor. (2017). https://en.wikipedia.org/wiki/C_preprocessor
[34] Polle T Zellweger. 1983. *An interactive high-level debugger for control-flow optimized programs*. Vol. 18. ACM, New York, NY, USA.