PyFront: Conversion of Python to C Extension Modules

Jonathan Riehl
United Space Alliance
July 16, 1998

Abstract

PyFront is a system for the conversion of Python modules into C extension modules. PyFront is related to the Paths static analysis tool for Python, but employs a higher fidelity data flow model. In building these higher fidelity models, PyFront bridges the gap between the interpreted Python language, and the compiled C language. The C extension modules generated by PyFront will provide faster execution times but identical results to the original Python source. PyFront has the potential to offer an intermediate, but automatic, step in the optimization of Python modules and routines.

Contents

Objectives

PyFront is designed to be a system for increasing the Python language's utility as a rapid application development tool. Using PyFront, working Python code can be translated into faster C code. The primary objective of PyFront is to minimize or eliminate any overhead added to execution times due to the interpretation of Python byte code and the frame stack. Additionally, PyFront has been designed with the following objectives in mind:

- Non-intrusive: The system should not require a developer to add any special code or create separate scripts for it to provide immediate benefit. If extra constructs were required or optionally available, they should not affect the operation of the original Python code input into the system.
- Compatible: The system should be syntactically compatible with the Python language. Any source module that was compatible with a given version of Python would be a valid input to the system. Furthermore, the system should not operate on syntactically incorrect source.
- Modular: The system should be designed to be used from the command line, allowing it to be incorporated into a build utility and/or an integrated development environment. The system should not require any

additional user interaction than an input Python file name.
- Optionally typed: The system should offer optional type constructs that would assist in the generation of optimized (faster and/or smaller) C code. The typing constructs should also provide better type safety, enabling the Python analysis phase to act in a similar fashion as the lint utility for C.

Origins

PyFront was conceived after some discussion of a "Python compiler" took place on the USENET. The brunt of the discussion was over the feasibility of a static Python compiler. Near the end of the thread, some had concluded that the most feasible approach to Python translation and analysis was something that was not quite a compiler, but not an interpreter (termed then as a "space potato.") The implicit use of such a system would be for generating faster executables from Python code.

The PyFront system is also related to work done for the GRAD and Paths systems that were under development during PyFront's inception [Fly]. PyFront borrows many of the objectives in the previous section from GRAD. GRAD, or Grammar-based Rapid Application Development, uses language grammars to drive automatic interface generation for extending Python. Acting like GRAD in reverse, PyFront uses Python's grammar to automatically build C extension modules. The Paths test automation system, a testing tool and direct descendant of the GRAD system, was designed to perform the majority of the analysis chores that PyFront uses. While Paths builds lower fidelity models (computation is abstracted to a set of "ideal" operations, causing the model to lose some accuracy,) it also served as a proof of concept for static data flow analysis of Python code. What would be required was the application of Paths technology to build higher fidelity models. These high fidelity mod! els would then be used in the gene ration of similar or equivalent C code, with the added bonus of code optimization, a by-product of the data flow model employed [Weise, et. al.]

Methodology

The primary methodology used in the development of PyFront consisted of analysis of the Python interpreter. Once the C basis of Python was understood, control and data flow models could be associated with Python operation. The most obvious model for control flow analysis centers around partitioning input code into basic blocks and building a control flow graph [Aho, et. al.] However, the data flow models used in prior systems would be more applicable to optimization, an intended use of the PyFront system. The following subsections discuss inquiry made into the Python interpreter's operation, as well various representations considered for use in Python translation.

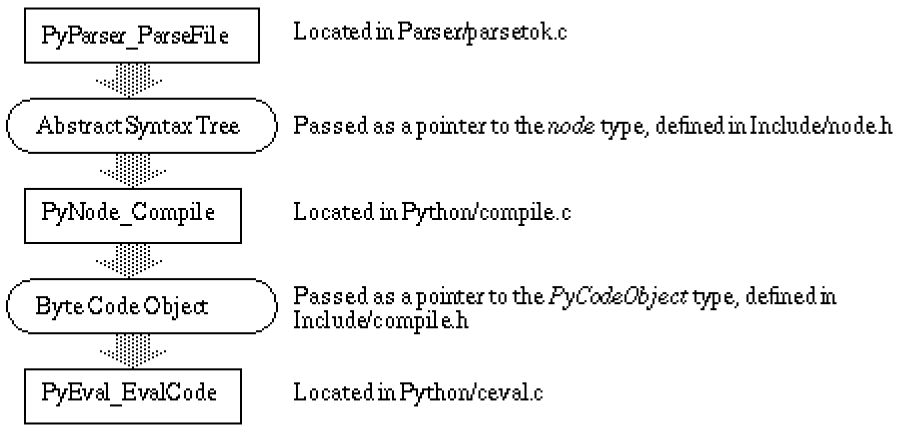

PyParser_ParseFile
Located in Parser/parsetok.c
Abstract Syntax Tree
Passed as a pointer to the *node* type, defined in Include/node.h
PyNode_Compile
Located in Python/compile.c
Byte Code Object
Passed as a pointer to the *PyCodeObject* type, defined in Include/compile.h
PyEval_EvalCode
Located in Python/ceval.c

Figure 1. A "Precise" Python Execution Model
(All file names are given relative to the base Python distribution directory.)

The Python Evaluation Loop

Figure 1 illustrates the exact routines used by Python and the intermediate data structures generated during the first time execution of a Python module (during subsequent executions, the byte code object is saved and reused, saving the time spent parsing the code.) A system to generate C code would need to mimic these routines, first generating an abstract syntax tree, then determine the byte code for the given syntax tree, and finally generate code that would carry out the sequence of instructions represented in the byte code.

One particular aid in the analysis of the interpreter process is the exposure of some of these routines in the Python standard library. In an introspective fashion, the parse tree and the byte code of most Python constructs may be examined in the interpreter itself. Specifically, the parser module allows the generation of abstract syntax trees, while the dis module allows Python byte code objects to be "disassembled" and their constituent byte code to be viewed on an instruction by instruction basis.

Since Python is capable of building abstract syntax trees, and handling byte code objects directly, all that remains to be analyzed is the code in the byte code interpreter. The interpreter (located in the Python/ceval.c source module) runs as a loop, handling byte code instructions one at a time.

Like a modern CPU, the interpreter loop keeps an instruction pointer, and has exception and error logic that halts script execution when a problem occurs. The evaluation loop isolates each instruction as a literal byte value and then takes an action based on the byte code encountered. The actions taken are to be found in a switch statement that references the entire Python byte code set as individual case statements. From these case statements are calls to the Python API [Van Rossum], with arguments to the functions either to be found in a stack or as an additional set of byte codes, following the operation's coded value.

Looking at the Python world from "in the loop" shows that as long as the Python data elements (which are always PyObject pointers in the C code) and byte code arguments are managed correctly, there is no difference between running the byte codes and making the Python API calls. If one were to walk Python byte code and emit the code found in the Python evaluation loop for the given code, there would be no difference to Python. The Python API was designed to easily interface with C code, allowing extension modules to be written in the faster C language, compiled into shared libraries and then imported into the Python interpreter.

Basic Block Analysis

There is a catch to an "evaluation loop inlining" strategy. There are byte codes that jump to different locations in the byte code string, not just the next byte code. There is a byte code to return from a function, returning control to a higher level Python function or the interpreter itself. There is an op code that raises exceptions which break out of the evaluation loop. Exceptions may also occur after any given API call, and require an extension routine in C to stop what it is doing, deinitialize any data being used in the function, and return immediately.

In order to handle these problems, the inlining routine would simply have to keep track of where the jump and raise instructions were (essentially partitioning the code into a set of basic blocks,) and where their targets were. In cases where the function being converted has no try-finally or try-except constructs, the following procedures would apply:

- Emit a function prologue, that will load the argument variables correctly.
- When an operation breaks from the evaluation routine, insert a goto instruction to an error handling section at the end of the function.
- When an unconditional jump is found, insert a label (such as "dest001:") before the code generated for the target instruction. Then emit a C goto instruction to the label inserted.
- When a conditional jump is found, insert a label at the code generated for its destination operation. Then emit a C "if" structure identical to the Python code, but replace the instruction pointer adjusting code with a goto to the label emitted.
- Emit an error and/or return section at the end of the function. The section(s) would be preceded by the label(s) referred to by any exits generated above. The section(s) would implement the exception and return logic found at the end of the Python evaluation loop.

The generated C code would take the interpreter logic and copy parts of it for each byte code instruction. It would only remove the necessity of regenerating and maintaining byte code (remember that this must now be handled by the developer using the inline processor and then a C compiler, which would be slower,) in addition to the minimal overhead of looping over the byte codes. Furthermore, it still ties us to the object stack, with inlined code pushing and popping values, unaware of the PyObject structures used by neighbor instructions (this is illustrated in figure 2 .)

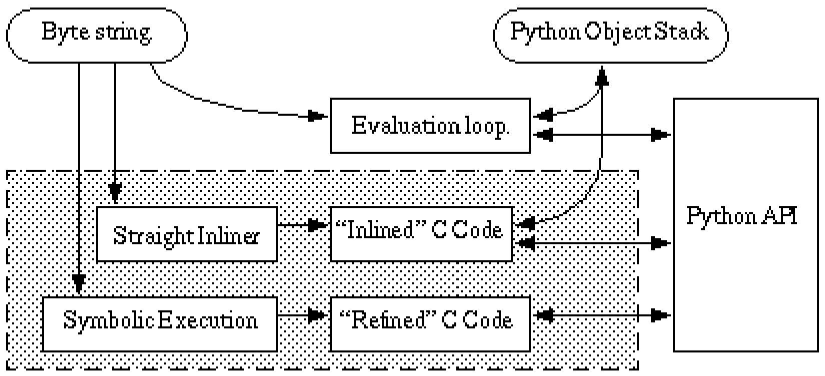


Figure 2. Various Data Flow Schemes for Python Compatibility

Data Flow Analysis

Figure 2 shows various aspects of the data flow within a Python session, in addition to depicting the "evaluation loop inlining" process described above. The diagram also shows a third process called "symbolic execution." Simply put, symbolic execution is a bridge between interpreter and compiler. Symbolic execution acts like an interpreter for parsed code, but instead of using real data, data flow is captured and place holders for real information are substituted. These place holders and their interconnection through the various operations in code, constitute a data flow graph.

An initial benefit of symbolic execution would be the removal of the stack from the generated code. The stack would not be required since symbolic execution, in order to create data flow graphs, must simulate the values being pushed on and popped off the stack. The principles of symbolic execution may be viewed in the following process:

- Partition the byte code into basic blocks. This is the process of tracking where the goto's and labels are emitted given the stratagem discussed above. The byte code between the labels and the goto's represent basic blocks.
- For each basic block, linearly handle the byte code, similarly to the Python evaluation loop.
- For each instruction, simulate the C code found in the Python evaluation loop. For function calls or even segments of the Python source that can not be simulated, generate data flow operations. For each pop in the Python C code, pop the data flow node off the simulated stack. If there is stack underflow, generate a data flow input node as a place holder. For each push, push the data flow node representing the generation of the value onto the simulated stack.
- At the end of a basic block, any data flow nodes remaining on the stack become data outputs for the block.
- Thread the inputs and outputs of basic blocks to each other. This is done using special nodes representing conditional entry, and re-entrant code containers.

An Example of Symbolic Execution

The above process is not trivial, but it may be more intuitively shown in the following example:

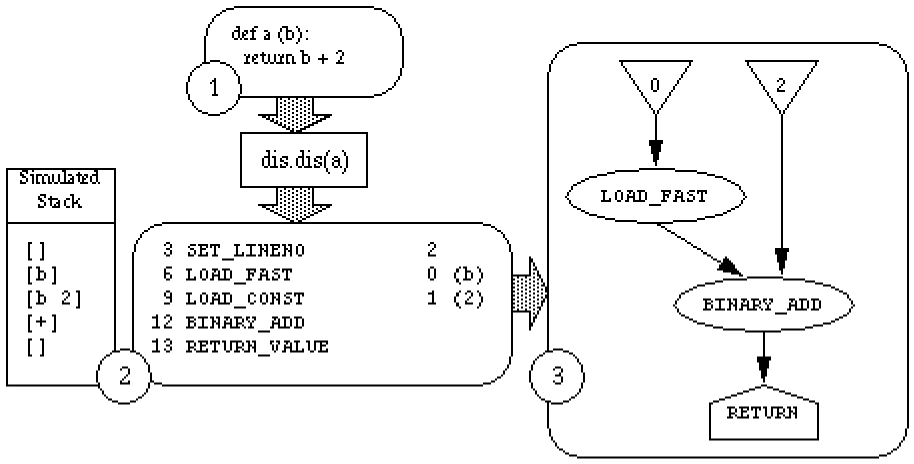


Figure 3. An Example of Symbolic Execution and Data Flow

Figure 3 shows an illustration of three things: a sample function (1), a byte code disassembly of the function (2), and a data flow graph (3). For the purpose of brevity, the sample function is very simple. There are no jumps in the function, and there is only one operation. Shown in the second area is an actual "disassembly" of the byte code Python generates for the given function. The byte code was obtained using the dis.dis routine, as illustrated in the flow. The final window shows a data flow graph generated from symbolic execution of the byte code, with two operations, two constants, and a value output.

To begin symbolic execution in this example, the byte code string for the given routine is analyzed (a function's byte code string is found in the co_code attribute of the actual function object's func_code attribute, or `a.func_code.co_code' in the example.) In this case, the byte code has no jumps, exceptions, and only a single return at the end (actually there is a second return that is automatically placed at the end of a function's byte code that will return None, but since this code will never execute, it was omitted from the disassembly in figure 3 .) Since the execution order of the entire function is linear, the byte code shown is the only basic block to be considered.

Shown beside the byte code listing for the example function is a stack list. The stack shown shows the result of symbolic execution of each byte code instruction. The following would occur during symbolic execution of the byte code:

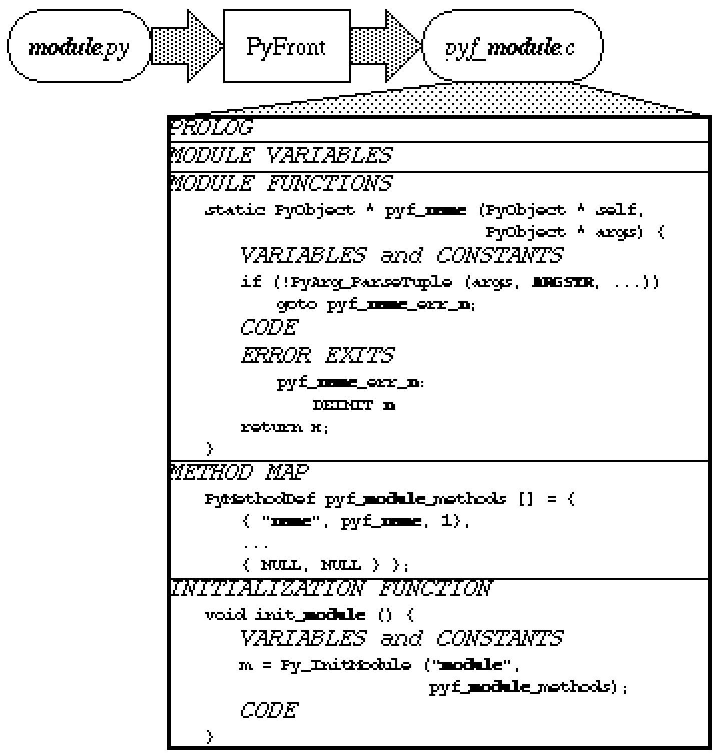
- The SET_LINENO op code is ignored in this example, but it could also be used during C code generation by setting up a special, sequential data flow representation. This would allow traceability when debugging the C code. (There is another SET_LINENO for line 1, but there is no code following it, so it was omitted from the op code listing.)
- The `LOAD_FAST 0' op code would reference the first value in the function's locals list, increment its reference count, and push the result onto the value stack. The function's locals list is initialized by the Python evaluation loop upon entry. In this case, the value passed for the `b' parameter would be present at that location. However, the value is not known during symbolic execution, so a `LOAD_FAST' data flow operation node is created, linked to its constant argument (0), and pushed on the stack (represented in the example illustration as `b', but the actual operation is pushed.)
- The `LOAD_CONST 1' op code would reference the second value in the function's constants tuple (referencing these in the `a.func_code.co_consts' value from Python shows that the first constant is None,) increment its reference count, and push the object onto the stack. Since the symbolic execution routine has the ability to reference the function constants, a constant data node for the number 2 is generated and pushed on the symbolic stack. This shows a very basic optimization achieved by symbolic execution: if a value is known at analysis time, it will be used by the system. The symbolic execution routine would remove the overhead required to index into the code object's constants and push any of them onto the stack.

- The `BINARY_ADD' op code would pop two values off the stack, perform some type checks to optimize integer addition, add the two values, decrement the reference count of the two popped objects, and push the result of the addition. Symbolic execution would pop the `LOAD_FAST' operation and the constant value node for 2 off the stack, create a `BINARY_ADD' operation node, build edges from the two data flow inputs to the addition operation, and push the `BINARY_ADD' node just created onto the stack.
- Finally, the `RETURN_VALUE' op code would pop the top of the stack, and break out of the evaluation loop, returning the object. Since a value is output here, the symbolic execution routine would pop a data flow node off the symbolic stack and build an edge from that node to a newly created `RETURN' node.

The data flow graph, with its inputs (downward pointed triangles) and outputs (upward pointed triangles) would then represent how data flows through the given function. At this point, the graph would be walked in a second pass of symbolic execution. The second pass would model value assignments and references. Ideally the second pass would extract such operations as `LOAD_NAME' and `STORE_NAME', by internally maintaining data flow aliases for stores, and building edges to these aliases for later load operations.

Code Generation

The end result of PyFront is a completed C extension module. Figure 4 shows the template used for the output extension code. The Python extension module is broken into four sections: the file prologue, the module functions, the method map, and the module initialization function. The models discussed in the methodology section are used to generate C code for module functions, and the initialization routine, while top level analysis is used to integrate the functions with the module name space and the method map.



```
PROLOG
MODULE VARIABLES
MODULE FUNCTIONS
    static PyObject * pyf_name (PyObject * self,
                                PyObject * args) {
        VARIABLES and CONSTANTS
        if (!PyArg_ParseTuple (args, ARGSTR, ...))
            goto pyf_name_err_n;
        CODE
        ERROR EXITS
            pyf_name_err_n:
                DEINIT n
        return x;
    }
METHOD MAP
    PyMethodDef pyf_module_methods [] = {
        { "name", pyf_name, 1},
        ...
        { NULL, NULL } };
INITIALIZATION FUNCTION
    void init_module () {
        VARIABLES and CONSTANTS
        m = Py_InitModule ("module",
                           pyf_module_methods);
        CODE
    }
```

Figure 4. A Template for Module Emission

The file prologue consists of a comment header, an include directive for the PyFront API, and a module constant section. The comment header simply identifies the source Python code, the version of PyFront used to generate the extension module, and the date of the conversion. PyFront generated C modules require that only the PyFront API header file is included. The PyFront API includes the Python API, and will be discussed later. Finally, a static array of Python object pointers is declared, and a constant initialization routine is generated.

In the Python interpreter, constants are instantiated during the byte code conversion process, and serialized as a part of the compiled module. PyFront initially generated C code that created constants during function initialization, and then destroyed the constants at the end of the function. The overhead was quickly determined to be undesirable, and constant handling was moved to the module level. All module constants encountered during PyFront's code analysis process are assigned an index in an array of Python object pointers. PyFront generates a static routine for instantiation of the constant array as a part of the module prologue. The constant handler routine is then called by the module initialization function.

After the module prologue, functions corresponding to function definitions in the source module are created. The C code for each function declares a set of Python object pointers, a call to the tuple argument handler, C code for the function, and code for handling various exit conditions. The function variables replace the object stack, the local variable array, and add the functionality of two registers. Then, the function arguments are loaded into local variables using the tuple parsing routine, and the function code is generated.

Function code is generated using symbolic execution of the byte code instructions. The byte code instructions may be divided into roughly four categories: control flow, processors, operations, and store operations. PyFront does not use the control flow op codes, but rather builds control flow models that are translated directly into C flow constructs. Such byte codes include the "JUMP_IF_FALSE" and "JUMP_FORWARD" instructions. The second category of instructions are processors. These op codes do not return a Python object, but rather return an integer error code. A good example of a processor operation is the "PRINT_ITEM" byte code. Next, there are operations. Operations will pop objects off the stack, perform a function on the popped objects, and push an object (or several) back on the stack. The "BINARY_ADD" instruction is an operation. Finally, there are store operations that provide interaction between the program name s! pace and the stack, or solely modi fy the stack. These operations include the load and store prefixed op codes and stack modifiers such as "POP_TOP."

During symbolic execution, instructions are dispatched to handler routines that simulate the instruction's stack actions, and build data flow representations. Currently, C code is generated at this phase of program analysis, generating roughly one function call in C for each instruction. Since several instructions make one or more Python API calls, and may involve special control logic, PyFront uses its own API. The PyFront API simply wraps the Python evaluation actions with some minor modifications made to stack dependent code, and includes the Python API. The PyFront API provides a single function for each Python op code, simplifying code generation, and reduces the size of the C extension code.

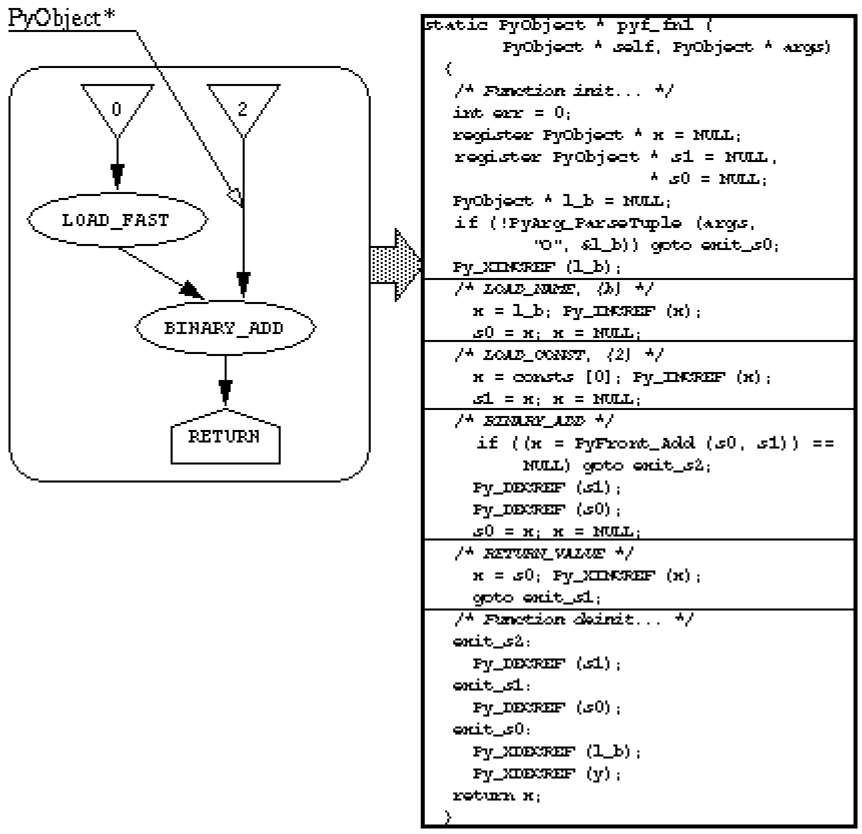
Error handling is done for each API call emitted, based on the type of the instruction. Processor instructions return a non-zero value to indicate an error, and are handled as follows:

```
if ((err = PyFront_Process (s0, ...)) != 0) goto exit_sx;
```

The goto is a reference to an exit label that is determined based on the simulated stack size. From the target exit handler on, the stack replacement variables are properly deallocated using the DECREF macro. Operation instructions return a Python object that must be non-NULL, hence a call to an operation instruction handler will appear as follows:

```
if ((x = PyFront_Op (s0, ...)) != NULL) goto exit_sx;
```

The object returned from the API call is placed first in a "register" variable, which is then moved to the stack simulator variables upon determination of a non-NULL result. Figure 5 shows an example of the C code generated by PyFront revision 0.4 for the demonstration code in figure 3 .



```
static PyObject * pyf_fn1 (
        PyObject * self, PyObject * args)
 {
  /* Function init... */
  int err = 0;
  register PyObject * x = NULL;
  register PyObject * s1 = NULL,
                    * s0 = NULL;
  PyObject * l_b = NULL;
  if (!PyArg_ParseTuple (args,
          "O", &l_b)) goto exit_s0;
  Py_XINCREF (l_b);
  /* LOAD_NAME, (b) */
    x = l_b; Py_INCREF (x);
    s0 = x; x = NULL;
  /* LOAD_CONST, (2) */
    x = consts [0]; Py_INCREF (x);
    s1 = x; x = NULL;
  /* BINARY_ADD */
     if ((x = PyFront_Add (s0, s1)) ==
           NULL) goto exit_s2;
    Py_DECREF (s1);
    Py_DECREF (s0);
    s0 = x; x = NULL;
  /* RETURN_VALUE */
    x = s0; Py_XINCREF (x);
    goto exit_s1;
  /* Function deinit... */
  exit_s2:
    Py_DECREF (s1);
  exit_s1:
    Py_DECREF (s0);
  exit_s0:
    Py_XDECREF (l_b);
    Py_XDECREF (y);
  return x;
 }
```

Figure 5. Emission of a Data Flow Representation

Results

The current implementation of PyFront is in its fourth revision (0.4,) and consists of a 5,500 line Python to C code generator, and a 1,000 line C extension API. This version of PyFront was informally benchmarked against Python version 1.5 for performance in simple iterative loops and recursive loops. The benchmarks were run on an IBM 43P 604e at 166 Mhz with 64MB RAM under AIX 4.2.1. The benchmark functions were implemented in a single 29 line Python module and converted to a 804 line C file (about 27.7 times larger.) In this case, PyFront was in a debug mode that generates comments about the corresponding Python code line number and op code (similar to those shown in figure 5.) The extension module generated was compiled using IBM C Set++ with the same optimization level as was used for the Python interpreter. Times were measured using the Python time module and averaged over ten trials.

While loop performance was first tested, using the following routine:

```
def while_test (i):
   while i > 0:
      i = i - 1
```

Figure 6 shows the results of executing the while_test routine over a range of iteration magnitudes. The data for a thousand iterations shows a very large maximum time trial. The varied data may be the result of sensitivity to system resources. Both Python and PyFront call the PyNumber_FromLong API routine extensively in this test. At numbers beyond 100, for the Python build used, memory must be allocated for the new integer objects being created by PyNumber_FromLong. In greater iteration sizes, the variance disappears as the iteration time is larger than the memory allocation overhead. Ultimately, the while loop proves to be PyFront's strong point, converging at speeds over 2.5 times faster than Python.

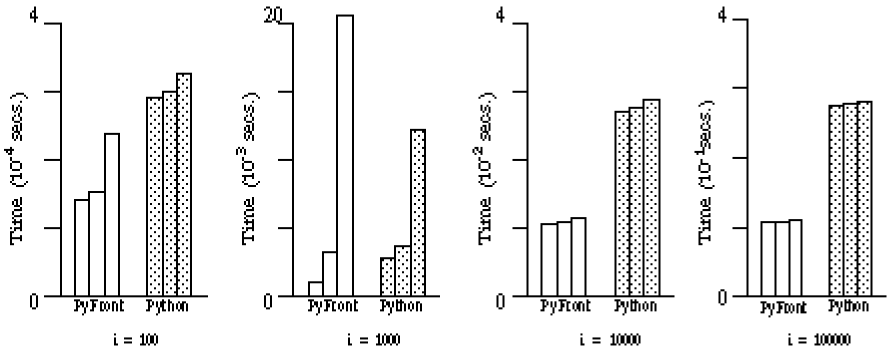


Figure 6. Results of while_test *benchmark for various iteration counts.*
(Columns represent minimum, average, and maximum times over ten trials.)

The next loop tested was the for loop, which employed the following routine:

```
def for_test (i):
   y = 0
   for x in xrange (0,i):
      y = y + 1
```

Figure 7 shows the results of running the for_test routine over the same number of iteration magnitudes used for the while loop. The PyFront implementation of "for" loop logic employs a comparable number of Python API calls to the interpreted version. The PyFront for loop shows a higher variance in timing data when compared to the while loop, and sometimes exceeds the Python measurements. Yet at higher iteration counts, the PyFront "for" loop converges at speeds roughly 1.6 times faster than Python.

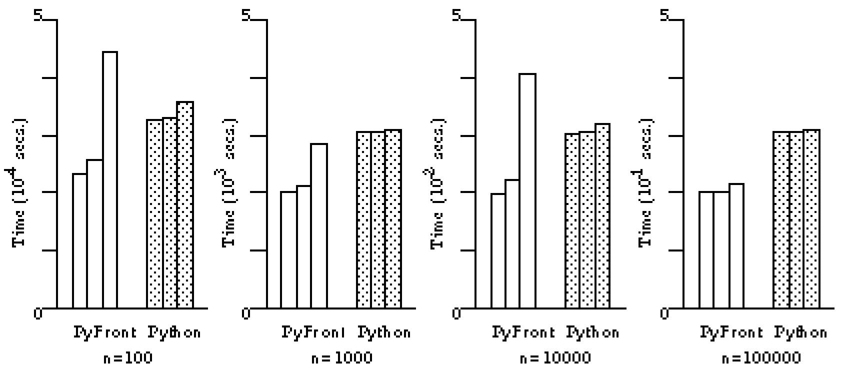


Figure 7. Results of for_test *benchmark for various iteration counts.*
(Columns represent minimum, average, and maximum times over ten trials.)

The final test used was a recursive loop, that ran the following function:

```
def recursive_test (i):
   if (i > 0):
      recursive_test (i - 1)
```

The data taken for the recursive trials was not available for higher recursive call sizes due to a frame stack limitation built into Python. Figure 8 shows that PyFront currently introduces a larger function call overhead, running over two times slower than the Python recursion trials. Recursion exposes an implementation trade-off in the current version of PyFront. PyFront builds a Python code object to wrap the generated C extension functions. By generation of a native Python wrapper, function arguments (specifically list and keyword arguments) are handled correctly, and a frame object is built, providing proper traceback information and global name space resolution. It also creates code that effectively makes two function calls instead of one (one to the Python wrapper function, and one to the C extension function. Note that calls from PyFront generated code to other Python functions do not add this overhead. The overhead only occurs for calls to PyFront generated functions.)

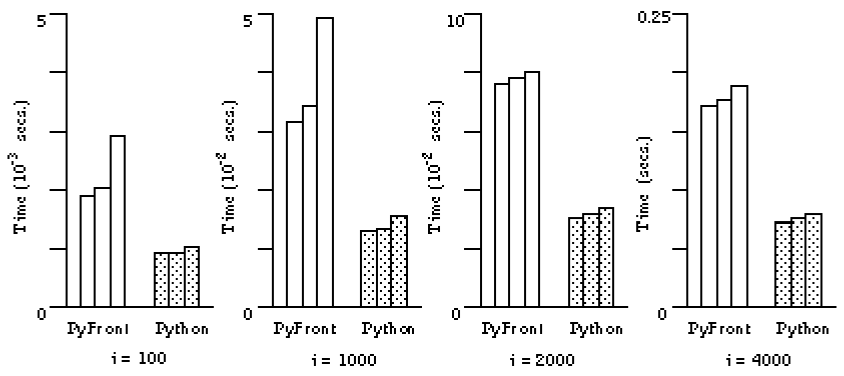


Figure 8. Results of recursive_test *benchmark for various recursion counts. (Columns represent minimum, average, and maximum times over ten trials.)*

Numerical results for the time benchmarks are available in the Benchmark Data section.

Future Directions

Since the current implementation of PyFront is only a prototype, future efforts will focus on building a system that fully meets the objectives of non-intrusion, compatibility, modularity, and optional typing. Specifically, the current prototype implements correct exception passing, but not exception handling (using the "try-except" and "try-finally" constructs.) The next goal is to increase the utility of PyFront as an optimization tool. Finally, as an enhancement to the optimization utility, a proposed typing system will be implemented and tested for compatibility and a low level of intrusion.

Completion of Python semantics will require that exception handling code be added to the extension modules generated. Complicating this objective, "return" and "break" statements may be interrupted by "finally" clauses before they exit their target subroutine or loop. Therefore, emission of "finally" clauses will require the code generated for "break" and "return" to jump to any "finally" clause rather than the exit point of a given loop or function. The "finally" exit point will require the use of an exit flag similar to the Python evaluation loop's "why" variable. The implications for symbolic execution mean that the frame stack must be simulated in addition to the value stack.

Once full compatibility is achieved, the second goal is to optimize the PyFront API and generated code. The current version of PyFront includes the use of local variables for the local name space (which was required to get loop times below Python 1.5 speeds,) and module wide constant initialization. Future optimizations may include constant folding, common sub-expression elimination, local variable aliases, and type based optimization. Constant folding would evaluate constant expressions, and generate the resulting constant instead of generating the Python API call. Common sub-expression elimination would use an operation hashing method to determine when a value may be reused, generating a temporary holding variable in the C code which would be used instead of performing redundant computation. Local variable aliases would simulate the local variable name space during symbolic execution, allowing any constant assignments to propagate through and be folded where appropri! ate.

Finally, type based optimizations will be prototyped in PyFront. The data flow graph built by PyFront provides a means for propagation of constant types, but would most likely be unable to consistently resolve the type of inputs and variables. PyFront would address the type resolution problem by providing a type extension scheme. The proposed type extension scheme would involve the insertion of a uniquely named dictionary at the top level of a module. The dictionary would map from regular expressions to strings containing type information. When a name is referenced during symbolic execution, the type system would attempt to match the label to one of the regular expression keys in the type dictionary. If type information was available, it would be used to select type specific operations, circumventing Python API calls. Otherwise, the standard Python API call would be emitted as already done by the current PyFront.

The PyFront type dictionary would offer the following features:

- Non-intrusive: The type dictionary would be parsed by Python as a normal module member. Specifically, the type dictionary would map a set of strings to strings, requiring no additional constructs, keywords, or import modules. The dictionary's presence would simply add some byte code overhead to the module, and would not change the module's semantics as a Python script.
- Self documenting: By using regular expressions as keys in the type dictionary, variable naming conventions could be employed in a similar fashion to currently used naming conventions (such as Hungarian prefix notation) [Simonyi.] These naming conventions, assuming some standard was followed, would increase code readability and type consistency even without the use of PyFront.
- Optional: The type dictionary would provide the opportunity to define types down to specific variables, but would not be required to use PyFront.

For example, Hungarian notation uses a "n" prefix for integers. In PyFront, adding a line such as "__pyf_type_dict = {`n[A-Za-z0-9_]*' : `int'}" would implement the integer prefix association. The prime drawback to this system (as any similar type system) is that legacy Python code would have to be modified to take advantage of type optimizations.

A final note about the type system: it could be used to express such concepts as static module variables (allowing global variables to be optimized in a similar fashion to local variables,) or even constance (allowing global constant assignments to be treated like macros.) If these type extensions were employed, recursive calls and calls to other functions in the module could be replaced by direct calls to the C function, eliminating the recursive call overhead seen in the Results section.

Conclusion

By translating Python control flow into C control flow, replacing the Python value stack with C variables, and inlining Python API calls normally made by the Python interpreter, PyFront achieves modest increases in Python script performance. While certain design trade-offs in the current implementation add overhead to calling converted functions, they are also needed to assist PyFront with its goal of compatibility. As these trade-offs are weighed and future plans are implemented, PyFront performance should continue to improve. Further development of such features as the proposed type system will enable PyFront to provide a non-intrusive, and mostly automatic path from Python modules to faster and statically compiled C code.

Works Cited

Aho, A.V., Sethi, R., and Ulman, J.D. *Compilers: Principles, Techniques, and Tools.* Computer Science Series. Reading, Massachusetts: Addison-Wesley, 1986.

Fly, Charles. "Grammar-based Rapid Application Development." http://www.python.org/workshops/1996-11/papers/GRAD/html/GRADpaper.book.html

Simonyi, Charles. "Program Identifier Naming Conventions." http://www.strangecreations.com/library/c/naming.txt

Van Rossum, Guido. *Python/C API Reference Manual.* Reston, Virginia: Corporation for National Research Initiatives, December 31, 1997.

Weise, D., Crew R., Ernst, M., and Steensguard, B. "Value Dependence Graphs: Representation Without Taxation." In Principles of Programming Languages, pages 297-310. Portland, Oregon: ACM Press, January

1994.

Benchmark Data

Results of while_test benchmark in seconds.

System	Average	Maximum	Minimum	i
PyFront	0.000151	0.000237	0.000140	100
Python	0.000299	0.000327	0.000295	
PyFront	0.003176	0.020530	0.001099	1000
Python	0.003690	0.012221	0.002732	
PyFront	0.010875	0.011622	0.010723	10000
Python	0.027499	0.028712	0.027156	
PyFront	0.108263	0.110256	0.106739	100000
Python	0.276231	0.282810	0.273740	
PyFront	1.081822	1.089260	1.078540	1000000 (not illustrated.)
Python	2.755389	2.828164	2.744902	

Results of for_test benchmark in seconds.

System	Average	Maximum	Minimum	i
PyFront	0.000257	0.000444	0.000234	100
Python	0.000330	0.000358	0.000326	

PyFront	0.002122	0.002833	0.002020	1000
Python	0.003060	0.003086	0.003050	
PyFront	0.022115	0.040743	0.019935	10000
Python	0.030748	0.031951	0.030324	
PyFront	0.203508	0.213573	0.200051	100000
Python	0.307384	0.309501	0.305754	
PyFront	2.019005	2.026224	2.012978	1000000 (not illustrated.)
Python	3.079594	3.130966	3.067530	

Results of recursive_test benchmark in seconds.

System	Average	Maximum	Minimum	i
PyFront	0.002010	0.002934	0.001886	100
Python	0.000949	0.001040	0.000936	
PyFront	0.034421	0.049501	0.031700	1000
Python	0.013491	0.015399	0.013080	
PyFront	0.078262	0.080017	0.076403	2000
Python	0.031356	0.033385	0.030542	
PyFront	0.177236	0.188143	0.171520	4000

Python	0.075773	0.080134	0.072615	