Aggressive Type Inference

John Aycock
Department of Computer Science
University of Victoria
Victoria, B.C., Canada
aycock@csc.uvic.ca

Abstract

Python is a ``dynamically typed'' language because, in general, the type of any variable is not known definitively until run-time. This feature is known to be a major limiting factor in optimization of Python code, and is typically addressed by calls for optional static typing to be added to Python. In this paper I describe an application for type inference unrelated to optimization, and present a new method for divining type information - aggressive type inference - which determines the types of variables in the absence of explicit cues. An empirical study of Python programs suggests that this might be a reasonable approach.

1 Introduction

In talking with people at last year's Python Conference (IPC7), I mentioned the possibility of writing a Python compiler... in Python. Not content to stop there, I suggested that the idea could be taken further, to translate Python code into Perl [29].

The idea of a Python-to-Perl translator has some merit. In fact, many of the arguments in favor of JPython [11] apply, particularly the ability to leverage Perl development. (And supply an alternative for Python programmers who are forced to work exclusively with Perl!)

Internally, both Python and Perl compile programs into code for a language-specific virtual machine (VM). This gives four avenues by which Python code may be translated into Perl, shown as dashed arrows in Figure 1. Some avenues are more promising than others, though. Both languages' VMs are fully specified in terms of concrete operational semantics [27], a polite way of saying that their details are buried in source code and subject to change. A translation involving either VM would result in unreadable and unmaintainable code.

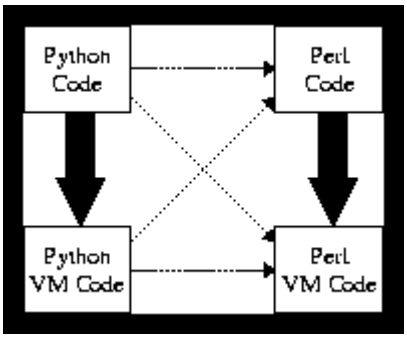

Figure 1: Python to Perl translation.

In contrast, a source-to-source translator would be ideal. Figure 2 shows a possible Perl translation for a snippet of Python code.

r

```
a = 123          $a = 123;
```

```
b = [a, 456]      @b = ($a, 456);
c = {'yyj': a}    %c = ('yyj' => $a);
print a,          print $a;
```

Figure 2: Translation example.

What does this have to do with type inference? The translated Perl code must have the `$@%&*` type specifiers on all Perl variables. This tells Perl, for instance, that a variable is scalar (string, number), an array, or a hash. Determining this type information is the task of type inference.

2 Type Inference

The process of inferring variables' types by looking at how they are used is called type inference.

Type inference has a long tradition in functional languages. Hindley[9] and Milner [19] independently discovered a method for inferring types at compile-time. Its most widely-known incarnation is in the language ML [18].

Type inference in functional languages is, in turn, based on work in the early 1960s on automatic theorem provers. In particular, Robinson [21] gave an algorithm for unifying logical expressions which was used later by Hindley and Milner for inferring types.

Algol-family languages [25], by comparison, have relied primarily on explicit type information supplied by the programmer, although there have been some attempts to the contrary [10].

These type inference systems are *conservative* in the sense that, given a variable X, they will always compute a superset of X's type [1].

3 Aggressive Type Inference

Python is a dynamically-typed language, meaning that the exact types of variables are not known until the program is run. In Figure 3, `x` is clearly an integer if S2 is executed, or a string if S3 is executed. And at S4? The exact type of `x` at S4 is indeterminate at compile-time, unless we know which part of the if-statement will be taken at run-time. In this case, that requires knowledge of `x`'s value, which cannot generally be known at compile-time.

```
def foo(x):
    print x       # S1
    if x:
        x = 123   # S2
    else:
        x = 'abc' # S3
    print x       # S4
```

Figure 3: Dynamic typing in action.

Even more problematic is the type of `x` at S1. If the program were to be analyzed in its entirety - whole-program analysis - we could attempt to locate all calls to `foo` and see what the possible types of `x` may be. This is a nontrivial task in itself. Code like `foo(17)` is easy to locate; discovering that `apply(functions[y+random()], (17,))` calls `foo` is undecidable.

The final complication is that Python, like many other scripting and functional languages, allows new code to be generated and executed at run-time. In the case of Python, this can be done directly with the `exec` statement, or more surreptitiously by creating a `.py` file on-the-fly and importing it. Type information

arising in this way is unobtainable at compile-time.

These problems are not unique to Python, and are known to implementors of other dynamically-typed programming languages. Work on Tcl compilers, for example, has universally noted the difficulty of type inference [14,22,23]. (The benefits are equally well known. Type information has been characterized as being critical to the efficient implementation of Smalltalk [13], SELF [5], and APL [17].)

How can types of Python variables be determined at compile-time? All proposed solutions to date involve (optional) static typing, which requires the programmer to explicitly insert type information. Variations on this theme include [16,20] and innumerable discussions on `comp.lang.python` and the Python Types-SIG.

I have taken a different approach with the idea of aggressive type inference (ATI). The key idea underlying ATI is this:

> *Giving people a dynamically-typed language does not mean that they write dynamically-typed programs.*

In other words, just because Python permits programmers to write code like that in Figure 3 doesn't mean that code like it is written frequently. A similar conjecture[1] about usage of Tcl variables is made in [14]; empirical evidence for this ``type consistency'' is presented in [28], where they found about 80% of operators in a large sample of Icon programs maintained the same type.

ATI works according to two rules:

1. Flow-insensitivity. This is a concept from data flow analysis. To quote Cooper and Kennedy [6],

   > `Flow insensitive information describes data flow events which occur on *at least one* path through a procedure... By contrast, flow sensitive information describes data flow events which occur on *every* path through a procedure.'

   Effectively this means that control flow is ignored. Applying this rule to Figure 3, ATI would decide that `x` has the type string Ènumber at S2, S3, and S4.

2. Type consistency within a scope. This second rule addresses the problem of determining `x`'s type at S1. If a variable has a type T during its lifetime, then it has type T at *every* point within the scope in which it is bound to a value. In Figure 3, ATI infers `x`'s type at S1 to be string Ènumber, because `x` has that type later in the function. Furthermore, ATI has reached this conclusion without having to look beyond the code for `foo`.

   An analogy can be drawn between this second rule and the scope rules in Pascal [12], where an identifier's scope is the entire block in which it is declared, not just from the declaration point onwards.

ATI has apparently resulted in some rather useless type information: no single distinct type for `x` has been arrived at! Now, suppose that a restriction is placed on Python programs. For a program to be used with ATI, it must be written in such a way that ATI can infer an exact type for all variables. Otherwise, a compile-time error will result. Figure 3 is an invalid program according to this restriction.

Of course, all Python programs do not adhere to this restriction, nor should they. This restriction is certainly acceptable for my Python-to-Perl application, though. It is in keeping with proposed ``high speed'' implementations of a Python subset, such as Swallow [7] and Viper [24]. And it is a similar restriction to that which optional static typing requires.

The above two ATI rules alone are insufficient to infer types for some programs. In Figure 4, looking at `c.set()` or `c.get()` in isolation does not allow any type inferences to be made. ATI can be used in

conjunction with other sources of type information, as I describe in the next section.

```
class c:
    def set(self, o):
        self.o = o
    def get(self):
        return self.o
```

Figure 4: Inference in isolation indeterminate.

4 ATI Implementation

I have developed a proof-of-concept implementation of ATI. Approximately 1200 lines of Python code, it operates in two phases (Figure 5):

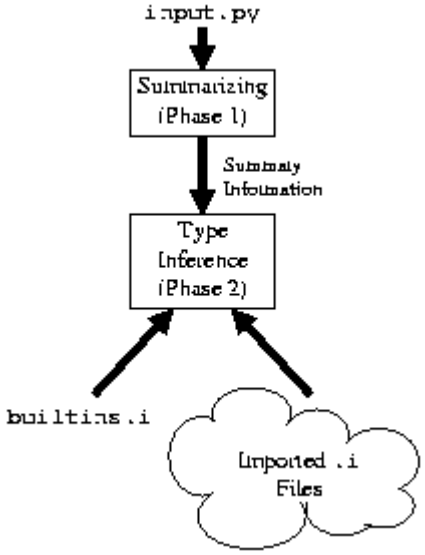

Figure 5: ATI Phases.

1. The input Python program is scanned and parsed. (No semantic checks are performed; the input is assumed to be correct.) All information relevant to ATI is distilled into summary information and saved into a file with a `.i` suffix. For example, if the code in Figure 4 resided in `blarg.py`, then its summary information would be written to `blarg.i`, whose contents are shown in Figure 6.

```
type c is class
scope c begin
    equ set.#1 = set.o
    equ set.#0 = set.self
    type set is func
    scope set begin
        assign #t4 = o
        equ self = #t3
        op #t4 is #t3 . o
    scope set end
    equ get.#0 = get.self
    type get is func
    scope get begin
        return #t2
        equ self = #t1
        op #t2 is #t1 . o
    scope get end
scope c end
```

Figure 6: Summary information (reformatted for legibility).

All information about control flow, such as branches and loops, is discarded. Information that *is* kept includes:

a. The scope of classes, methods, and functions (`scope`).
b. Variable assignments (`assign`).
c. Operations on variables (`op`).
d. Method/function return types (`return`).
e. The types of names (`type`).
f. Equivalences between names (`equ`). In Figure 6, for example, it is noted that the zeroth parameter to `set` is an alias for `set`'s local variable `self`.
g. Import statements (`import` - not shown).
h. Global declarations (`global` - not shown).

2. The summary information is repeatedly examined in order to propagate type information. For example, given the summary information

```
assign x = y
assign y = #t1
type #t1 is string
```

ATI would discover on the first pass that there are three names: `x`, `y`, and `#t1` (a temporary name generated by phase 1). It would also note that `#t1` has the type *string*. On the second pass, ATI would find that `y` has the type *string* too. Finally, ATI would conclude on the third pass that `x` is a *string*. (This process can be made much more efficient!)

The names of imported modules appear in a file's summary information. When this is encountered in phase 2, an attempt is made to read a `.i` file for the imported module. It is not an error if such a file is missing: in this way, ATI can be given either partial or whole-program information, as appropriate.

As a general rule, ATI will be more effective the more information it is given. Taking the code from Figure 4 as input, my ATI implementation only decides that `c` is a *class*, `self` is an *instance*, and `set`/`get` are *method*s. But given the few extra lines of input highlighted in Figure 7, the correct types of all names in the input are inferred:

`c`	class
`c.set`	method
`c.get`	method
`c.set.self`	instance
`c.get.self`	instance
`c.set.self.o`	number
`c.get.self.o`	number
`x`	instance
`y`	number

```
class c:
    def set(self, o):
        self.o = o
    def get(self):
        return self.o
```

```
x = c()
x.set(123)
y = x.get()
```

Figure 7: More information, better inference.

Type information for Python's built-in functions is automatically imported from the file `builtins.i`. An excerpt from its source file is shown in Figure 8; these functions are defined skeletally because only type information is required.

```
def abs(N):
    N = 123
    return 123

def dir(object=None):
    return ['abc']

def range(a1, a2=123, a3=123):
    return [123]
```

Figure 8: Some built-in function definitions.

Both phases use the little language framework described in [2]. I am currently reworking the ATI implementation to overcome some design limitations in Phase 2. In particular, inference involving *list*s is incomplete, making full analysis of real programs troublesome.

5 On Being Wrong

ATI may arrive at a solution which appears to discover types for all variables, yet is incomplete in the sense that all possible types for variables have not been found. Furthermore, it is not generally possible to detect this situation. Consider the following cases:

1. Only partial information is given to ATI. In this case, ATI can obviously miss vital information.

2. Code is generated at run-time. Since ATI is done at compile-time, it cannot be privy to run-time information. This may be mitigated to some extent by warning about uses of the `exec` statement, and imported modules for which no information is available at compile-time.

3. The call graph is obscured. By this I mean code where the flow of control is unclear. For example, function calls made through an array of function pointers, or a class which dynamically changes its superclass.

In these situations, the ``aggressive'' nature of ATI comes into play. I assume that even though cases such as the above are possible, that they occur infrequently and thus may be ignored.

6 Applicability

It is a rather bold claim to say that Python programs, overall, are not especially dynamic. I have done some static and dynamic analysis of programs which suggest that there is some truth to this claim. (Only the results are presented here; the details of the analyses are deferred to the Appendix.)

Seven bodies of Python code were chosen for this survey, a total of 51,300 lines of code (LOC):

1. Idle 0.4, a graphical user interface for Python development bundled with the Python 1.5.2

distribution.
2. Gadfly 1.0, a relational database system [30].
3. Grail 0.6, an Internet browser [4].
4. HTMLgen 2.2, a generator of HTML documents [8].
5. J--, a compiler for a subset of Java.
6. Lib, the Python 1.5.2 library (sans subdirectories).
7. Pystone 1.1, a Dhrystone benchmark included in the Python 1.5.2 distribution.

The code was statically analyzed for indications of code being dynamically generated and executed: the `exec` statement, and uses of `eval()`, `execfile()`, and `__import__()`. (These are cases where ATI will fail.) The results are shown in Table 1, broken down by individual constructs in addition to being taken as a total percentage of lines of code. [2]

	`exec`	`eval`	`execfile`	`__import__`	LOC	% of LOC
Idle	1	0	1	1	4449	0.07
Gadfly	0	2	0	1	10200	0.03
Grail	4	2	0	0	6419	0.09
HTMLgen	0	4	1	2	4794	0.2
J--	0	0	0	0	1498	0.0
Lib	11	23	2	12	23754	0.2
Pystone	0	0	0	0	186	0.0

Table 1: Static occurrence of dynamic constructs.

Not surprisingly, the static occurrence of these four constructs is highest in Python's library code, where esoteric code would be expected. But even in the library, the frequency of these constructs is insignificant when compared to the number of lines of code. However, as the Python library code is clearly atypical, and it is not obvious how to fairly test such disparate components, I excluded it from further direct analysis.

For dynamic analysis, each package was run using an instrumented Python interpreter. This interpreter counted the dynamic occurrence of the same four constructs and the number of Python VM instructions executed. These results are shown in Table 2. Again, the results suggest that dynamic code generation facilities in Python are not heavily used.

	`exec`	`eval`	`execfile`	`__import__`	Instructions	% of Instructions
Idle	1	0	0	12	346617	0.004
Gadfly	0	47	0	0	7957055	0.0005
Grail	214	6	0	0	4676698	0.005
HTMLgen	0	831	10	0	422496	0.2
J--	0	0	0	0	8096543	0.0
Pystone	0	0	0	0	6702077	0.0

Table 2: Dynamic occurrence of dynamic constructs.

Having few impediments to compile-time analysis of Python code is important; for ATI, having variables that don't change from one type to another is even more important. Using the instrumented interpreter again, I tracked VM instructions that store values to local and global variables, classifying each instruction's effect on a variable's type into one of three categories:

1. T_{NULL} -> T_X. No change of type, because the variable has no prior value (this happens when a variable hasn't been assigned to previously, or has been deleted with `del`).

2. T_X -> T_X. No change of type resulted.
3. T_X -> T_Y. A change of type resulted.

	T_{NULL} -> T_X	T_X -> T_X	T_X -> T_Y	Stores	T_X -> T_Y (as % of Stores)
Idle	12462	8827	1633	22922	7.1
Gadfly	482644	185353	7444	675441	1.1
Grail	107154	99651	8410	215215	3.9
HTMLgen	10755	11351	835	22941	3.6
J--	67475	1238553	1820	1307848	0.1
Pystone	220247	290005	7	510259	0.001

Table 3: Dynamic type consistency from VM store instructions.

	Frame Objects			*Code Objects*		
	Total	T_X -> T_Y	%	Total	T_X -> T_Y	%
Idle	6705	1285	19.2	369	28	7.6
Gadfly	164196	3261	2.0	792	61	7.7
Grail	78413	7472	9.5	1252	124	9.9
HTMLgen	11594	795	6.9	579	24	4.1
J--	50174	1814	3.6	284	11	3.9
Pystone	170106	6	0.004	61	6	9.8

Table 4: Localization of type changes.

Table 3 shows the results of this experiment. Obviously, the vast majority of VM store instructions do not result in a type change. This raises several questions.

Are T_X -> T_Y localized? In other words, are type changes made throughout the program, or are they confined to small portions of code? Table 4 shows that there are only a few culprits. (Roughly speaking, frame objects correspond to function/method invocations, and code objects correspond to functions/methods in the program text.)

What does this mean in terms of real variables? The data in Table 3 does not tell the whole story in terms of variables in the program text. For example, the code

```
x = 123
del x
x = 'abc'
```

would appear as two T_{NULL} -> T_X stores, rather than a T_X -> T_Y. Another extreme case would be where every variable's type is changed once, then remains constant thereafter. Since ATI is dependent on the type consistency of variables in the program text, I used the same dynamic data to reconstruct the local variables in each code object. As the results in Table 5 show, the numbers in Table 3 are not misleading.

	Total Local Variables	Locals with > 1 Type	% with > 1 Type
Idle	1801	20	1.1
Gadfly	3143	150	4.8
Grail	4668	91	1.9

HTMLgen	1581	15	0.9
J--	801	2	0.2
Pystone	264	0	0.0

Table 5: Dynamic type consistency of local variables.

Are certain type changes predominant? The short answer: no. This seemed to depend heavily on the particular application; in J--, for example, over 99% of the type changes were made from `None` to some other type. This predominance was not true in general, however.

7 A Tale of Two Type Systems

The original motivation for ATI was Python-to-Perl conversion, determining the appropriate Perl type specifiers for converted Python variables. How can ATI be applied?

In an apparent contradiction, Figure 3 would convert easily into Perl - the resulting program is shown in Figure 9. This, despite the assertion in Section 3 that ATI must infer an exact type for variables! The catch is that Python-to-Perl conversion involves *two* type systems. To be more precise, ATI must infer a type for each Python variable which has a mapping to a unique member of the type system of interest.

```
sub foo {
    my ($x) = @_;
    print "$x\n";    # S1
    if ($x) {
        $x = 123;    # S2
    } else {
        $x = 'abc'; # S3
    }
    print "$x\n";    # S4
}
```

Figure 9: Dynamic typing: the resulting Perl.

For example, Figure 10 shows a mapping between ATI-inferred types and Perl types. Figure 3's code is acceptable by this mapping because both Python strings and numbers map into Perl scalar variables. (This particular mapping was used for a manual translation of some Python code into Perl.)

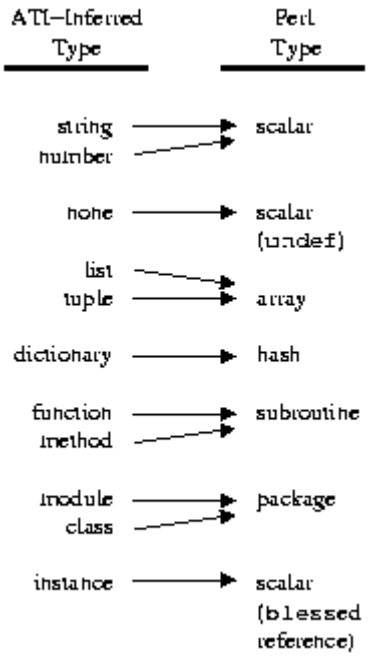

Figure 10: Mapping to Perl types.

Another example with two type systems would be a Python optimizer which would inline string operations, and would therefore want to locate all variables with a string type. Here, the inferred types would have to map into the types *{string, not-string}*.

8 Future Work

There are many applications which can benefit from type information. Progress on a Python-to-Perl translator is now possible using the ATI information; type checkers and optimizers also rely on type information. A tool to convert legacy Python code to an optional static typing scheme might be interesting too.

ATI may benefit by embracing and extending type inference research done for other dynamically-typed languages. This includes taking control flow into account [26], and adding run-time checks to convert programs into a form that can be type-checked at compile-time [3].

Dynamic ATI would be a logical variation of this work. A modified Python interpreter could record type information as a program runs, over multiple runs, later inferring types of variables based on the run-time information.

I would also like to extend the empirical type analysis in Section 6, using a wider sampling of Python programs. This would further gauge the amount of type consistency present in Python programs, and would give a good indication of the general applicability of ATI.

9 Conclusion

Type inference is a difficult task for dynamically-typed languages such as Python. The type information gathered, however, is essential for some applications, such as the Python-to-Perl translator I proposed.

By making aggressive assumptions about how programmers use variables in their programs - namely, that variables maintain a consistent type throughout - it is possible to make type inferences that would not be possible with a more conservative approach.

With aggressive type inference, I have demonstrated how type inference may be done in Python without requiring the programmer to supply explicit type information.

Acknowledgments

Shannon Jaeger and Jim Uhl proofread this paper and made many helpful suggestions, as did the anonymous referees and Jeremy Hylton. Also, thanks to Roger Jaeger and the University of Calgary Department of Computer Science for use of their computer equipment while I was travelling. Nigel Horspool suggested mapping store instructions to their corresponding variables. This work was supported in part by a grant from the National Sciences and Engineering Research Council of Canada.

Appendix

In this section, I present the methods used for static and dynamic analysis, so that the results are repeatable and may be extended or reinterpreted.

Static analysis of Python code was performed by a program which lexically analyzed all `.py` files in a package using the `tokenize` module. Comments and blank lines were ignored, so the ``lines of code'' calculated for each package is accurate.

All `NAME` tokens reported by `tokenize` were examined for the names `exec`, `eval`, `execfile`, and `__import__`. Conceivably, use of the latter three could be cloaked by assigning them another name, but this would be questionable programming practice and was deemed unlikely. Another possible problem would be code which re-used the name of a built-in function for a different purpose; this would artificially inflate the static occurrence counts. In fact, this is exactly what happened with Gadfly, which re-used the name `eval`, necessitating manual analysis.

Other program constructs could prove detrimental to compile-time analysis too, such as `apply()`, `setattr()`, and manipulations of `__dict__`. A more sophisticated static analysis is required to determine if an occurrence of one of these other constructs would be problematic.

For dynamic analysis, a task was chosen for each package which was intended to exercise a reasonable subset of the package's code. This proved difficult to gauge for graphical applications like Idle and Grail; a future study should employ code coverage tools. The chosen tasks are shown in Table 6.

Idle	Loading `testcode.py`
Gadfly	Running `gftest.py`
Grail	Loading `http://www.python.org/`
HTMLgen	Running `HTMLtest.test()`
J--	Generating MIPS assembly for a recursive-descent calculator
Pystone	Full execution

Table 6: Tasks for dynamic analysis.

The Python interpreter was modified to log a number of events, including:

1. Frame object allocation and deallocation, and their corresponding code objects.
2. Calls to the built-in functions `eval()`, `execfile()`, and `__import__()`.
3. Execution of the `IMPORT_NAME` instruction. Since this instruction calls `__import__()`, the data reported for `__import__` in Table 2 was adjusted by the number of times `IMPORT_NAME` was executed. This way, only the explicit calls to `__import__()` are significant.
4. Execution of the `EXEC_STMT` instruction.
5. Execution of the `STORE_NAME`, `STORE_FAST`, and `STORE_GLOBAL` instructions, the target variable, and the type of that variable before and after the instruction.
6. Instruction execution, for instruction counting.

The generated log files were later processed to produce the dynamic data reported in Section 6.

References

[1]
A. Aiken and B. Murphy. Static type inference in a dynamically typed language. *Proceedings of the 18th ACM POPL*, 1991, pp. 279-290.

[2]
J. Aycock. Compiling Little Languages in Python. *Proceedings of the 7th International Python Conference*, 1998, pp. 69-77.

[3]
R. Cartwright and M. Fagan. Soft Typing. *Proceedings of the ACM PLDI '91 Conference*, 1991, pp. 278-292.

[4]

Corporation for National Research Initiatives. Grail. http://grail.cnri.reston.va.us/grail/.

[5]

C. Chambers. *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*, Ph.D. Dissertation, Stanford University, 1992.

[6]

K. D. Cooper and K. Kennedy. Efficient Computation of Flow Insensitive Interprocedural Summary Information. *SIGPLAN 19*, 6 (June 1984), pp. 247-258.

[7]

M. Faassen. Re: The way to a faster python [was Python IS slow !] Posting to `comp.lang.python`, May 1999.

[8]

R. Friedrich. HTMLgen. http://starship.python.net/crew/friedrich/HTMLgen/html/main.html.

[9]

R. Hindley. The Principal Type-Scheme of an Object in Combinatory Logic. *Transactions of the American Mathematical Society 146* (December 1969), pp. 29-60.

[10]

O. I. Hougaard, M. I. Schwartzbach, and H. Askari. Type Inference of Turbo Pascal. *Software: Concepts and Tools 16* (1995), pp. 160-169.

[11]

J. Hugunin. Python and Java: The Best of Both Worlds. *Proceedings of the 6th International Python Conference*, 1997.

[12]

K. Jensen, N. Wirth, A. B. Mickel, and J. F. Miner. *Pascal User Manual and Report (Third Edition)*, Springer-Verlag, 1985.

[13]

R. E. Johnson, J. O. Graver, and L. W. Zurawski. TS: An Optimizing Compiler for Smalltalk. *OOPSLA '88 Proceedings*, 1988, pp. 18-26.

[14]

B. T. Lewis. An On-the-fly Bytecode Compiler for Tcl. *Proceedings of the Fourth USENIX Tcl/Tk Workshop*, 1996.

[15]

B. T. Lewis. Private communication. September, 1999.

[16]

R. E. Masse. Evolutionary Prototyping: ``Add Later" Static Types for Python. *Proceedings of the 7th International Python Conference*, 1998, pp. 91-101.

[17]

T. C. Miller. Type Checking in an Imperfect World. *Proceedings of the Sixth ACM POPL*, 1979, pp. 237-243.

[18]

R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[19]

R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences 17* (1978), pp. 348-375.

[20]
J. Riehl. PyFront: Conversion of Python to C Extension Modules. *Proceedings of the 7th International Python Conference*, 1998, pp. 79-90.

[21]
J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM 12*, 1 (January 1965), pp. 23-41.

[22]
F. R. Rouse and W. Christopher. A Tcl to C Compiler. *Proceedings of the Third USENIX Tcl/Tk Workshop*, 1995, pp. 115-122.

[23]
F. R. Rouse and W. Christopher. A Typing System for an Optimizing Multiple-Backend Tcl Compiler. *Proceedings of the Fifth USENIX Tcl/Tk Workshop*, 1997.

[24]
J. M. Skaller. RFC: Viper: yet another python implementation. Posting to `comp.lang.python`, August 1999.

[25]
R. Sethi. *Programming Languages: Concepts and Constructs*. Addison-Wesley, 1989.

[26]
O. Shivers. Data-Flow Analysis and Type Recovery in Scheme. In *Topics in Advanced Language Implementation*, P. Lee, ed., MIT Press, 1991, pp. 47-87.

[27]
K. Slonneger and B. L. Kurtz. *Formal Syntax and Semantics of Programming Languages*. Addison-Wesley, 1995.

[28]
K. Walker and R. E. Griswold. Type Inference in the Icon Programming Language. TR 93-32a, University of Arizona Department of Computer Science, 1996.

[29]
L. Wall, T. Christiansen, R. L. Schwartz, and S. Potter. *Programming Perl (2nd Edition)*. O'Reilly, 1996.

[30]
A. Watters. Gadfly. `http://www.chordate.com/gadfly.html`.

Footnotes:

[1]Unfortunately, unconfirmed as of this writing [15].

[2]The `eval()` number for Gadfly was determined by manual inspection, for reasons discussed in the Appendix.

File translated fromTEXby TTH,version 2.56.
On 23 Nov 1999, 19:45.