LOCALIZED TYPE INFERENCE OF ATOMIC TYPES IN PYTHON

A Thesis
Presented to
the Faculty of the California Polytechnic State University
San Luis Obispo

In Partial Fullfilment
of the Requirements for the Degree
Masters of Science in Computer Science

by
Brett Cannon
June 2005

LOCALIZED TYPE INFERENCE OF ATOMIC TYPES IN PYTHON

by

Brett Cannon

APPROVAL PAGE

TITLE: LOCALIZED TYPE INFERENCE OF ATOMIC TYPES IN PYTHON

AUTHOR: Brett Cannon

DATE SUBMITTED: May 24, 2005

Dr Aaron Keen

_____ _____

Advisor or Committee Chair Signature

Dr Gene Fisher

_____ _____

Committee Member Signature

Dr Michael Haungs

_____ _____

Committee Member Signature

Abstract

LOCALIZED TYPE INFERENCE OF ATOMIC TYPES IN PYTHON

by

Brett Cannon

Types serve multiple purposes in programming. One such purpose is in providing information to allow for improved performance. Unfortunately, specifying the types of all variables in a program does not always fit within the design of a programming language.

Python is a language where specifying types does not fit within the language design. An open source, dynamic programming language, Python does not support type specifications of variables. This limits the opportunities in Python for performance optimizations based on type information compared to languages that do allow or require the specification of types.

Type inference is a way to derive the needed type information for optimizations based on types without requiring type specifications in the source code of a program. By inferring the types of variables based on flow control and other hints in a program, the type information can be derived and used in a constructive manner.

This thesis is an exploration of implementing a type inference algorithm for Python without changing the semantics of the language. It also explores the benefit of adding type annotations to method calls in order to garner more type information.

Acknowledgements

Special thanks must be given to the Python language development team. Not only did they help give the author his (serious) start in programming in general and languages specifically, but have also been very supportive of this work.

The Python Software Foundation deserves indirect thanks for funding the PyCon 2005 Python community conference. The PyCon 2005 paper committee deserves direct thanks for allowing the data collected for this thesis to be presented at PyCon 2005.

Friends and family deserve thanks for listening to my constant ramblings about this thesis, even when they had no clue what was being told to them.

Finally, thanks must also be given to Dr Aaron Keen for acting as thesis advisor to the author. His continual support and positiveness, even when the negative results became apparent, has always been greatly appreciated.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Types play an important role in programming. Types can be used to check that illegal operations do not occur between disparate values. Type information can also be used to improve performance, such as identifying integers and performing integral math at the assembly level [5].

Typically types are statically declared in order to provide information to the compiler. Unfortunately, declaring types is not always feasible. The programming language being used may not support specifying types for variables, as is the case in most dynamic languages. "A dynamic language is one that defers as many decisions as possible to runtime" as defined by Guy Steele. Constantly specifying types is viewed as superfluous by some languages and thus not supported. Type inference provides the ability to gain type information without requiring type declarations.

Type inference is the process of finding the most accurate type information for a program that does not explicitly state the types of variables at compile-time, all without inferring any inaccurate information [1]. Type inference allows one to

have the benefits of type information at compile-time without the programming overhead of explicitly specifying types.

Consider the simple Standard ML function:

```
fun add2 x = x + 2;
```

There are no type declarations specifying the argument types or the return type of the function. Using type inference, though, Standard ML infers that the type of the function is "`int → int`" (the notation is to be read as all types up to the last one are arguments to the function, with the last value being the return type). The argument is inferred to be of type `int` because the `+` operator requires `int` operands. The addition expression is the only one in the function, making the type of the expression the return type of the function, which is `int`. The programmer is able to program type-safe code without statically specifying types and to gain a performance increase from the type information inferred.

Not all languages that lack type declarations have type inference as a step in their compilation. Python is a dynamic, open source programming language [23]. Python does not have any type declarations. In terms of type information provided to the compiler, Python's compiler only knows of atomic types (i.e., syntactically supported types) and only at the point of creation. Type information is not propagated through Python's compiler for future use.

This thesis explores whether more type information at compile-time from type inference would benefit Python. Specifically, we explore if introducing type inference into Python's compiler along with type-specific bytecodes for Python's interpreter will lead to at least a 5% performance increase[1] across various benchmarks without any semantic changes to the compiler or language.

[1]The 5% goal has been chosen since it is an informal rule of thumb used by Python's development team as a measurement of whether something is worth the added code complexity.

In addition to introducing type inference, optional type annotations for function and method parameters have been introduced into Python (for the rest of this paper, the term "method" will represent both functions and methods for simplicity). This work analyzes the introduction of type annotations as a mechanism for providing more type information to the type inference algorithm. This is done because optional static type checking had been proposed as a possible addition to the language (as of this writing, though, that is no longer the case [20]).

Unfortunately the goal is not met. As will be covered in this thesis, the 5% performance is not reached for any of the benchmarks used for measuring the effectiveness of the type inference algorithm and the new bytecodes.

This thesis is laid out as follows: Chapter 2 explores the two primary type inference algorithms in use today. The challenge of applying a type inference algorithm to Python is covered by Chapter 3. In Chapter 4, other attempts at performing type inference upon Python are covered. The algorithm developed for this thesis is discussed in Chapter 5. The new bytecodes developed for a performance increase are covered in Chapter 6. In Chapter 7, the results of benchmarks are revealed. Chapter 8 discusses the conclusion reached for this thesis. Finally, Chapter 9 covers possible future work based on the result of this thesis.

Chapter 2

Type Inference Algorithms

2.1 Overview

There are essentially two primary type inference algorithms in use today. The Hindley-Milner algorithm is mostly used in functional languages and was introduced in Standard ML[15]. The Cartesian Product algorithm is geared towards object-oriented languages and was first used in the Self programming language [2].

2.2 Hindley-Milner

The Hindley-Milner algorithm (created by Robin Milner, basing his work on earlier findings of J. Roger Hindley, and implemented in Standard ML [4]) works with polymorphic types. A polymorphic type acts as a variable that represents any possible type. The identity function illustrates this well:

```
fun identity x = x;
```

Standard ML infers the function to be of type "'a → 'a". The function will accept any value and will immediately return it. This is represented by the fact that 'a is a polymorphic type. When a call to identity is made, 'a is set to the monomorphic (i.e. specific) type of the argument and any type-specific work is done based on the type 'a is set to.

Polymorphic types are not the only types inferred in the Hindley-Milner algorithm. As shown by the add2 function example in Chapter 1, and its inferred type of "int → int", monomorphic types are inferred when enough information is present.

2.2.1 Example

For a more thorough example, consider the following function:

```
fun foo x y =
    if x = y
        then x - y
        else x + y
;
```

Using a bottom-up implementation of Hindley-Milner (the algorithm can be implemented top-down, called the M algorithm, or bottom-up, called W [9]), foo's type is inferred to be "int → int → int"[1]. The algorithm takes five steps to reach the result.

First, all function parameters, x and y in the example, are type inferred to polymorphic types, 'a and 'b. No type information is known about how the function parameters are to be used, leading to the most broad type possible to be inferred.

[1] Standard ML supports function currying. The example can be read as "a function that takes an int that returns a function that takes an int and returns an int.

Second, the `if` guard, `x = y`, is type inferred. Comparisons in Standard ML are only legal between the same types. Type inferring the `=` operator forces `x` and `y` to be of the same type, expressed by both `x` and `y` being assigned type `'a`.

Third, the true branch is type inferred. With the operator being `-`, the argument types are required to be equivalent. The guard of the `if` statement handled the equivalency restriction. For mathematical operators, Standard ML's type inference algorithm forces polymorphic types to `int`. `'a` then becomes `int`. `x` and `y` both become `int`.

The fourth step is to infer the types of the false branch. The `+` operator works like the `-` operator for type restriction. No change is required in the types of `x` and `y`.

The final step is to infer the types of the return type. The `if` statement is the only expression for the function. Both true and false branches contain expressions whose return type is `int`. The `if` statement's requirement of the types of both branches being equivalent is met. The return type of the `if` statement is thus `int`.

2.2.2 Limitations

Hindley-Milner has some limitations which have been overcome through research [11]. One limitation is that types can only be inferred to be a monomorphic type or polymorphic type. A set of acceptable types cannot by inferred, leading to types having to be either exact or overly broad.

Second, higher-order polymorphism is not supported [11]. Higher-order polymorphism is "the use or definition of functions that behave uniformly over all type constructors" [11]. Subsequent research has fixed this limitation.

Lastly, polymorphic arguments that are used as different types in different locations in a method are not allowed [11]. Consider the example:

```
fun notallowed x y =
    if y > 0
        then x * 2.0
        else x * 2
;
```

The function argument `x` is used as a `real` in the true branch of the `if` statement and as an `int` in the false branch based on the value of `y`. An error is raised by Standard ML since `x` cannot be type inferred as both `int` and `real`. Both uses are possibly legal depending on `y`. Researchers have not solved this issue with Hindley-Milner.

2.3 Cartesian Product

2.3.1 Overview

Technically the Cartesian Product algorithm performs type inference for method calls only. Another algorithm, called iterative type analysis, infers the type of the body of methods [5]. Iterative type analysis creates a constraint graph of types for the body of a method and performs three steps to reach its conclusion.

First, type variables are allocated for all of the variables and constants in the body of a method. The type variables are the nodes of the constraint graph. The nodes represent a single monomorphic type. Polymorphic types are not used by iterative type analysis.

Second, these type variables are seeded with their initial types. A base type that all types derive from is used when no initial type is known.

Third, constraints are created by drawing edges between the nodes and the operations performed on the nodes. What types a variable may be is determined by following the edges of the graph until it terminates. The set of all visited nodes represents all monomorphic types that the variable may be. The graph models program execution using types. The algorithm can be viewed as an abstract interpreter executing based on types [24].

There is another algorithm called the control flow analysis algorithm which works in a similar fashion to iterative type analysis [5]. Control flow analysis has the limitation of not allowing control flow to change while inferring types. The Cartesian Product as implemented in [2] did change control flow during type inference . This led to iterative type analysis being used.

2.3.2 Example

Consider the example Python program:

```
def foo():
    x = 42
    y = x * 3.5
    x = []
```

For the first statement, nodes for `x` and `42` are created. `x` is given the base type, `object`, and `42` is given the integral type. Edges are created from `x` and `42` to the assignment operator. Following the edge from `x` to the assignment operator joined to `42` shows the type of `x` to be the integral type.

The second statement creates nodes for `y` and `3.5`. `y` is seeded with the `object` type and `3.5` is given the `float` type. Edges are created from `x` and `3.5` to the * operator. An edge from the * operator to `y` through an instance

of the assignment operator is created. Based on promotion rules in Python, `y` is considered to have the type `float`.

The last statement creates the node for `[]`. The node is seeded with the type `list`. An edge is drawn from `[]` to the assignment operator which is connected to `x`.

Following all of the edges connected to `x` leads to the set of {`list`, integral} types, derived from the assignment nodes connected to `x`. `y` has the type `float` (the possible type derived from `[] * 3.5` because of the assignment of `[]` to `x` is ignored because it would be an error in execution and `x` has a valid result from the assignment of `42`). The final diagram is shown in Figure 2.1.

2.3.3 Execution Flow

Execution flow has a direct influence on type inference. Take the block of code:

```
y = 1
y = True
```

The first statement creates nodes for `y` and `1`. After seeding they are connected to each other through the assignment operator. The second statement adds the `True` node which is seeded with `bool` and connected to `y` through another instance of the assignment operator.

How execution flow is handled changes the types that `y` is inferred to have. There are two ways to handle execution flow; in a flow-insensitive or a flow-sensitive manner.

With flow-insensitivity, `y` has edges to all operations that influence its value in all parts of the body of code it is used in. `y = 1` and `y = True` lead to `y` having edges to the `1` and `True` nodes. The inferred type for `y` is the set {integral, `bool`}.

Flow-sensitivity rewrites the constraint graph at each statement, when possible, to more accurately match execution flow. The edge from `y` to the assignment operator connected to `1` is removed when the edge from `y` to the assignment operator that is connected to `True` is created. The inferred type for `y` is `bool` at the end of the code block. The result is more accurate than is found with a flow-insensitive type inference.

2.3.4 Method Calls

For inferring the types of method calls the key issue is minimizing memory usage and processing time while still being as accurate as possible. Method calls connect to a node called a template in the constraint graph. A template is a mapping from method arguments to the return type of the method call based on the argument types. Older algorithms created individual templates for each method call. This approach wastes memory and processing time for the same method call when the same argument types are recalculated at every point of use in the constraint graph. To minimize waste, a select number of templates were generated by other algorithms. Accuracy is sacrificed due to the limitation of templates [5].

The Cartesian Product algorithm solves the limitations of older algorithms by creating the Cartesian product of all possible argument types for a method call and storing the result in a single template. The template can be viewed as a hash table for the method call, keyed on monomorphic argument types and with the

value of the return type of the call. Recalculation and wasted memory is evaded by lazily adding new argument type keys to the template. Accuracy is improved by not limiting the number of argument type combinations the template can hold values for.

2.3.5 Accuracy

The Cartesian Product algorithm when coupled with iterative type analysis is accurate. The limitations that befall Hindley-Milner do not hold for the Cartesian Product algorithm. The drawback of the Cartesian Product compared to Hindley-Milner is implementation complexity; Hindley-Milner is known for being a fairly straightforward algorithm to code [11].

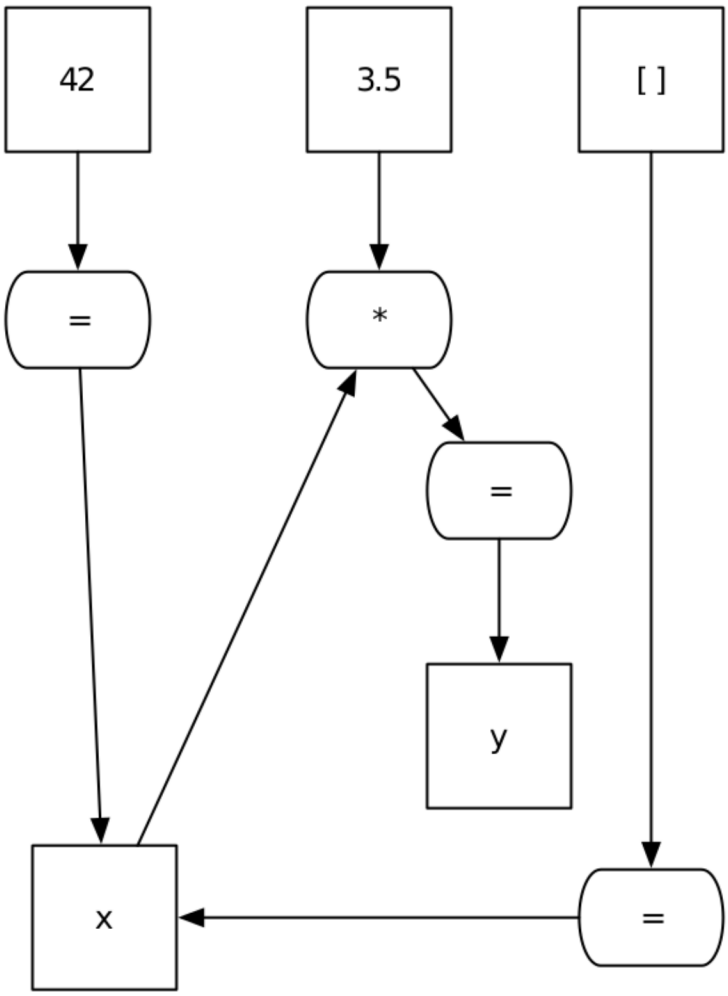

Figure 2.1. Graph of Cartesian Product Example

Chapter 3

Challenges of Inferring Types in Python

None of the algorithms mentioned in Chapter 2 can be directly applied to Python code. Type inference requires complete information on the control flow of an application [2]. Any lack of control flow information leads to a possible code path that introduces different types that could invalidate the inferred types.

In order to get complete control flow information all type information or source code must be present at compile-time [19]. For interpreted languages, the need for complete flow control information at compile-time does not necessarily exist.

3.1 Lack of Compile-Time to Run-Time Code Integrity

3.1.1 Structure

The goal of this work is to infer types in Python without any changes to the semantics of the language. Python's compiler is a part of the language's interpreter and is the second step in code execution (the first being parsing the source file). When a Python file (often ending in a ".py" extension) is passed to the Python interpreter, a check is done for a file with the same name but with a ".pyc" extension. If the ".pyc" files exists, it is assumed to be a bytecode-compiled version of the file being executed. Python's bytecodes are practically type-ignorant; only the specific bytecodes for creating atomic types (i.e. syntactically supported types) have any indication of types and they only specify what is going to end up on the interpreter's stack [22].

If a ".pyc" file does not exist, the input source file is parsed, compiled, written to a ".pyc" file, and then executed. Direct access to the compiler is possible through Python's standard library, but it is not a standalone application from the interpreter itself.

The compiler operates in an isolated fashion. It is passed a parse tree from the parser and emits Python bytecodes based off of that parse tree. There is no outside referencing of data or other source code. These operational semantics are permitted by the language definition based on the allowance of execution of code not present during compile-time.

3.1.2 Problems Caused

The lack of integrity of code from compile-time to run-time eliminates the possibility of garnering any control flow information from outside the current module. When the compiler reaches an `import` statement[1] it emits the proper bytecode to perform the importation at run-time. There are no checks for what will be imported at run-time.

The lack of code dependency checks at compile-time removes any guarantee that the code compiled against is the same as that used at run-time. In Python it is legitimate to compile against one module but use another one that is entirely different (except in name) at run-time. As long as the code executes without error there will be no direct difference apparent to the code using the module. Assume someone compiles against a module containing the following code:

```
import UserList

def debug_append(self, item):
    '''Print out what item is appended to the UserList'''
    print 'appending', item
    UserList.UserList.append(self, item)

def zero_to_ten():
    '''Return a list containing the numbers from 0 to 10, inclusive'''
    list_of_numbers = UserList.UserList()
    list_of_numbers.append = debug_append
    for num in range(11):
        list_of_numbers.append(num)
    return list_of_numbers
```

This example module contains a method that, when called, returns a new list containing the integers from 0 to 10, inclusive. The method, though, returns a `UserList` which mimics the list API of the `list` atomic type while being fully

[1] The `import` statement pulls code into the current module's namespace from other modules.

implemented in Python. The example code includes debugging information that would be unnecessary to include in the final code. One might do a final version as:

```
def zero_to_ten():
    '''Return a list containing the numbers from 0 to 10, inclusive'''
    return range(11)
```

The above returns a `list` unlike the version that uses `UserList`. To any code using the objects, there is no discernible difference between the two implementations unless you probe the returned object for type information. It is entirely plausible one could use the `UserList` version during development for the extra debugging information, and thus compile against it, but then release the `list` version to the public all without recompiling. There is no way to guarantee that the code used at run-time is the same used at compile-time.

Python's compiler could be augmented to check that the code being used at run-time is the same as that used at compile-time. Unfortunately, this would change the semantics of the language. Currently the only type of verification performed on ".pyc" files is that the version of the bytecodes used in the file is executable by the interpreter and, if the source file is also present, that the timestamp matches the original source the ".pyc" file is based on to keep the ".pyc" in sync with the corresponding ".py" file. But, if the source file is not present, the timestamp check is skipped. Execution of ".pyc" files that do not have the original source file present is allowed. Requiring the source file to be present to check that the code present at compile-time is the same at run-time would change the semantics of the language significantly compared to how it operates now.

3.2 External Modification of the Global Namespace

The lack of a guarantee that the code present at compile-time is the same as at run-time is not the only obstacle for inferring types in Python code. The language's dynamic execution, which is considered one of its strong points, poses its own set of issues.

3.2.1 Structure

Python's dynamic nature allows great flexibility in what can be done at run-time. The flexibility of Python extends to namespace manipulation between modules. In Python, one is allowed to inject values into another module's global namespace, regardless of whether this action shadows a name in the built-in namespace or replaces an existing value. Consider the following code:

```
'''In module shadowed.py'''

answer = 42

def answer_to_life():
    print answer
```

When executed `shadowed.answer_to_life()` prints `42`. But if you run the following code:

```
'''In module cause_trouble.py'''

import shadowed

shadowed.answer = -13

shadowed.answer_to_life()
```

`-13` is printed by `shadowed.answer_to_life()`. The code in `cause_trouble` is able to directly manipulate the global namespace of `shadowed` without the code in `shadowed` knowing something was changed by external code. The ability to manipulate namespaces also extends to shadowing names in the built-in namespace; one could shadow the built-in `len()` method, for instance, with a method that always returned `1` in `cause_trouble` with the line `shadow.len = lambda ignore: 1`.

3.2.2 Problems Caused

Python's allowance of external change to the global namespace proves to be a severe hindrance when paired with the issues caused by the lack of code integrity between compile-time and run-time as discussed in Chapter 3.1.2. It is not possible to rely upon the control flow of code in either external code to the code under current compilation or control flow accessed from the global namespace within the current module.

3.3 What Is Possible

There is some control flow that is guaranteed to not change between compilation and execution regardless of changes to the global namespace. Everything defined in the local namespace is known to be reliable from compile-time to run-time. Since there is no directly supported feature of Python that allows changing the control flow of code declared within a function or method, this can be considered reliable and essentially static (one can get direct access to the bytecodes of a function or method before execution and modify it, but the language does not guarantee that any changes you make will lead to semantically valid code [23]).

Only knowing about control flow at the local namespace level severely restricts what types can be considered while performing type inference. Only atomic types can be type inferred. Because atomic types are hard-coded into the actual source code there is, therefore, no worry of them being changed from compile-time to run-time. Classes cannot be considered during type inference since they may change between compile-time and run-time.

Compile-time knowledge cannot, however, extend to what is contained within atomic container types. In Python there are three atomic types that act as containers; dicts, lists, and tuples. The first is a hash table type, lists are mutable arrays, and tuples are immutable arrays. Because tuples are immutable, what they contain can be considered consistent. But both dicts and lists are mutable, making information about the values they contain unreliable.

While the language does not directly support an easy way to gain access to mutable arguments defined in the local namespace, the ability still exists through the `sys` module in Python's standard library and the method `sys.settrace()`. The function sets a trace method that is called when a method call is made, a new line of code is about to be executed, a method returns, or an exception is raised [22]. Triggering the trace method `before` the execution of any line of code allows one to change a mutable object just before its use. The following code, for instance, will replace all values contained within locally defined lists and dicts with `None` before access to the objects is allowed in subsequent code:

```
from sys import settrace

def replacewithNone(frame, event, arg):
    '''Replace all values contained within locally declared dicts and
    lists with None.

    Ignoring the 'event' value since this code is meant to be as
```

```
    vicious as possible and thus do not care about what the specific
    event triggered the trace function.

    '''
    for val in frame.f_locals.itervalues():
        if type(val) == dict:
            for key in val.iterkeys():
                val[key] = None
        if type(val) == list:
            for pos in xrange(len(val)):
                val[pos] = None
    return replacewithNone

settrace(replacewithNone)
```

The types that can be safely inferred in the local namespace are ints, longs, ASCII strings, Unicode strings, floats, complex numbers, lists (the type itself, nothing contained within a list), dicts (with the same restriction as lists), and tuples (the types of contained items can be type inferred if their types could be inferred outside of a container).

Chapter 4

Previous Attempts to Infer Types in Python

In spite of all of the difficulties in inferring types in Python code, others have tried. Two projects specifically are known to have tried two very different approaches to type inference in Python. Neither provides a solution that satisfies the challenge posed by this thesis.

4.1 Psyco

Psyco is a just-in-time (JIT) compiler that is a re-implementation of the main eval loop for Python [18]. It tries to detect ints and strings that are consistent from compile-time to run-time. Armed with this information, the new eval loop emits x86 assembly to perform calculations on those variables.

Psyco infers locally defined ints and strings directly and does not modify any other types. Unfortunately all of its work is done outside of the compiler and thus

does not provide an answer to the question posed by this thesis as to whether Python's compiler can stand to benefit from inferring types.

4.2 Starkiller

Michael Salib, while a graduate student at the Massachusetts Institute of Technology, created Starkiller, a type inference tool for Python source with an eye for eventual use as a tool to help transform Python source code into C++ code [19]. Using the Cartesian Product algorithm, it attempts to infer types in Python source code within certain limitations (such as not inferring when `eval()` or `exec` are present).

Ignoring its handful of limitations, Starkiller did a complete type inference of Python source. Its flow-insensitive algorithm is able to infer entire programs. Unfortunately, in order to infer types, Starkiller ignores the language semantics discussed in Chapter 3.2. The tool assumes that the code used while inferring types is the same as that used during execution, thus eliminating the ability to use different code at run-time than is used at compile-time.

Chapter 5

Inferring Atomic Types in the Local Namespace

In Python one can infer atomic types created in the local namespace without changing the semantics of the compiler or language. But just knowing this fact does not make it happen. One must develop an algorithm and an implementation of that algorithm to make this information useful.

5.1 Algorithm Overview

The algorithm designed to infer atomic types in the local namespace is almost the exact same algorithm as defined for iterative type analysis [5]. This is purely coincidental; the algorithm was independently developed before the iterative type analysis algorithm was discovered for this thesis. This independent development bodes well for the algorithm in terms of effectiveness.

For any type inference algorithm to work in Python, all possible control flow

constructs must be handled by the algorithm, as stated in Chapter 3. Conditionals are in the form of `if` statements. Looping is through `for` and `while`. Finally, exception handling changes control flow through `try/except` and `try/finally` statements.

The algorithm is flow-sensitive. As mentioned in Chapter 2.3.3, the inferred type of a variable can change as control flow is followed to a more restrictive type. Some circumstances do require flow-insensitivity instead of flow-sensitivity, such as when a change in control flow may occur at any point, and so a flow-insensitive style is also supported and used as needed. A global variable in the implementation signals which style is to be used. This restricts the algorithm to only using one style at a time.

In terms of atomic types, the algorithm can handle integrals (considered either `int` or `long` since in Python there is supposed to be no significant difference), `float`, `complex`, `basestring` (base class representing either `str` or `unicode`), `list` (delineated with square brackets, `[]`), `tuple` (delineated by parentheses), and `dict` (delineated with curly braces; `{}`). With a finite set of possible types for any locally defined variable, a bit set is used to represent the set of types.

To handle the actual type inference calculations, an abstract, types-only interpreter that mirrors how Python's stack-based interpreter works was written. As bytecode is emitted by the compiler it is passed to the type interpreter. Performing type inference during bytecode emission removed the need for a separate phase during compilation.

5.2 Conditionals

The initial types of variables upon reaching a conditional statement are used when execution reaches a branch of the conditional. Inside a branch the types of variables are viewed in isolation and thus can be as accurate as possible without worrying about which types a variable might hold in another branch of the same conditional. Upon exiting the conditional, the types for variables are inferred to be the union of all types that the variable can hold upon exiting any branch. A union is used over other set operations, such as intersection, because the inference must reflect `any` type a variable may hold while being as tight a bound as possible.

Here is a toy example:

```
var = 1

if var == 2:
    var = []
elif var == 3:
    var = {}
else:
    var = 'hi'
```

`var` is of integral type when the `if` statement is reached. To infer the types of the conditional, the initial guard is reached with `var` still inferred to be of integral type. In the branch for the initial guard, `var` is inferred to be of type `list`. The algorithm is flow-sensitive at this point and thus the inferred type for `var` is overwritten by assignment. For the `elif` guard `var` is initially inferred to be of integral type since each possible branch can be viewed in isolation thanks to the guarantee that only one branch will be taken during execution. At the end of the code block in the `elif` branch, `var` is inferred to be of `dict` type. Finally, the `else` branch is like the other guards upon entry; `var` is initially of integral type. By the end of the `else` branch `var` is inferred to be of `basestring` type.

The inferred type of `var` is `list` at the end of the initial branch, `dict` at the end of the `elif` branch, and `basestring` at the end of the `else` branch. After the execution of the `if` statement, the inferred type of `var` will be the set {`list`, `dict`, `basestring`}. We do not know how actual execution will go through the conditional thus `var` may be of any type along a path through the conditional.

Another view is that each branch body is type inferred in isolation from other branches in a flow-sensitive manner. The inferred types for the entire statement after its execution is the union of all the types from all branches.

5.3 Looping

Looping poses an interesting challenge since a variable will have its initial set of inferred types when it enters a loop body, but this set can change in the loop body and thus differ when it passes through the loop on the next iteration. Thus, compilation decisions cannot necessarily depend on information inferred on the initial pass through the loop.

Here is an example of a `while` loop:

```
var = {0 : 'hi'}
cnt = 0

while cnt < 2:
    var[0]
    var = ['hi']
    cnt += 1
else:
    var = 'hi'
```

The inferred types of `var` is initially `dict` upon entry into the body of the loop. The first execution of `var[0]` is expected to be executed on a `dict` based

on the inferred type up to that point. But `var` is inferred to be a `list` based on the next line. Continuing execution, the loop body is executed again, but `var` holds a `list` at this point. The second execution of `var[0]` is against a `list`, not against `dict` as initially determined. The initially inferred type of `dict` was incorrect for use on `var[0]`.

Since it is possible for the initially inferred type to be wrong, one cannot necessarily base compilation on this type. The inferred types of variables after a loop body are those resulting from a flow-insensitive type inference. In the example the type inference would lead to the set {`dict`, `list`} for the main loop body.

Because the implementation infers types during bytecode emission, bytecodes can be emitted that are incorrect because of the initial type inference. A check is performed after every looping statement to see if types have change for any variable that existed before entry into the loop body. If a change is detected, the bytecodes emitted for the loop body are tossed and compilation is performed again with the proper inferred types. If loops are nested, recompilation might be repeated but it will be superfluous.

Inference of the `else` branch is flow-sensitive. In the example, the inferred type of `var` in the `else` branch is `basestring`. If a `break` statement is detected, the type inference is flow-insensitive to handle the possibility of the `else` clause not being executed.

`break` and `continue` statements do not influence the type inference for loops. The flow-insensitive type inference takes into account the possible change in execution flow caused by a `break` or `continue` statement by incorporating all possible types for variables.

`for` statements act the same as `while` statements and thus will not be explicitly covered. A `for` statement differs from a `while` statement only in the inferred type (`object`) of the loop variant.

5.4 Exception Handling

Python supports exception handling through `try/except` and `try/finally` [23]. The former allows one to specify which exceptions are to be caught within a `try` statement and the code to use to handle the caught exception. For the latter, the code in the `finally` block is executed regardless of whether an exception is raised or not. You cannot mix `except` and `finally` branches in the same `try` statement.

The execution flow of the body of a `try` statement is is not guaranteed to be sequential. Any statement within the block can raise an exception. Without modeling the complete control flow of the application it is unknown which exceptions might be raised. Thus, every line must be considered a possible cause of any exception.

For example, here is a `try/except` statement:

```
var = 1

try:
    var = []
    var = 'hi'
    var = []
except:
    var = {}
else:
    var = ()

var
```

The `try` block is entered with the types that variables held upon entry, integral for `var` in this case. A flow-insensitive type inference of the `try` block gives the inferred type of `var` as the set {integral, `list`, `basestring`}; integral from the type upon block entry, `list` from the two `list` assignments, and `basestring` from the assignment of `'hi'`. Upon exit, all `except` blocks are examined as if the `try` block was analyzed in a flow-insensitive manner.

The `except` branch has the types set to what they are upon exit of the `try` block. Type inference is flow-sensitive since after execution of the `except` branch control flow exits the `try/except` statement. In the example, the type for `var` is inferred to be `dict` for the `except` branch.

The `else` branch works much the same as the `except` branch. Because execution flow exits the `try/except` statement after the execution of the `else` block, a flow-sensitive analysis is sufficient. `var` is type inferred to be `tuple` for the `else` branch.

The inferred types after the `try/except` statement is the union of the types for all `except` branches and the `else` branch, if present. In the example, this leads to `var` having {`dict`, `tuple`}. If an `else` branch is not present, then the type for the `try/except` statement is the union of all `except` branches and the types at the end of the `try` block. The change in type inference is due to the fact that the `else` statement acts as a guarantee that at least one branch will be taken out of the `try` block if no exception is raised. Had the example lacked an `else` branch it would have `var` inferred to the set {integral, `list`, `basestring`, `dict`}.

For type inference, a `try/finally` statement can be considered the same as a `try/except` statement with no `except` branches and the `finally` branch

acting like an `else` branch. Where an exception might be raised in the `try` block of a `try/finally` is unknown and thus requires the block to be analyzed flow-insensitively. The `finally` branch is guaranteed to be executed and thus is analyzed in a flow-sensitive manner.

5.5 Type Annotations of Function Parameters

As a proof of concept, and for initial experimentation, type annotations for methods were introduced by hand and embedded into the documentation string of the method to specify the type of every argument. Actual arguments were not checked against these specified types because there is no way to at compile-time and thus had to be correct else lead to an incorrect type inference.

5.6 Static Type Checking

The type interpreter also performed minimal static type checking of arguments of the methods of the atomic types. The methods of the atomic types cannot be changed by Python code and thus are considered static and can be taken into consideration during type inference. The return types of methods were also taken into consideration.

Static type checking did not turn up any errors in any of the thoroughly tested code run against the modified compiler, but it did properly detect a handful of errors in Python's regression test suite that were designed to do the same checking for the errors at run-time.

Chapter 6

Type-Specific Bytecodes

Having all of this type information at compile-time is useless in terms of performance if we cannot somehow harness it to speed up execution. It was decided that type-specific bytecodes would be introduced to increase performance. To choose which bytecodes should have type-specific variants, a group of nine projects were analyzed as listed in Table 6.1.

Project	Version	Lines of Code	Reference
BitTorrent	3.4.2	7,026	[6]
Mailman	2.1.5	21,099	[25]
Python Imaging Library	1.1.4	16,118	[17]
Plone	2.0.4	62,086	[14]
Pyrex	0.9.3	10,912	[8]
PythonCard	0.8	36,813	[3]
SciPy	0.3	65,042	[7]
Twisted	1.3.0	110,211	[12]
Python Standard Library	2.3.4	129,814	[22]

Table 6.1. Projects Used for New Bytecode Selection

The compiler was modified to output which types were being used for all

bytecodes as well as which methods were being called on the atomic types and the argument types used for the call. In instances where order of arguments to a specific bytecode was superfluous, the types were sorted lexicographically to eliminate any difference. If the type of the argument was not important it was left out.

Statistics emission was done at compile-time. Run-time was not considered because it would not be an accurate reflection of what the compiler would be able to infer at compile-time when decisions based on type information are made.

Two separate rankings were calculated to facilitate new bytecode selection. The first was the frequency of each bytecode/argument-type or method/argument-type detected, listed in Table 6.2. `STORE_SUBSCR` with a `dict` argument has such a high use because of how Python creates a `dict` defined syntactically; an empty `dict` is created and pushed on to the execution stack with repeated calls to `STORE_SUBSCR` to seed the `dict` with its values. `BINARY_MODULO` with a `basestring` has a high frequency because the modulo operator being overloaded for `basestring` to perform string interpolation; `'Hello, %s' % 'World'` will return the string, `'Hello, World'`. The rest of the bytecodes have a more gradual decline in usage.

The second ranking was based on the frequency of the data compared to the number of lines of code (LOC) to give a ratio of bytecode/method to LOC[1], listed in Table 6.3. Both `STORE_SUBSCR` with a `dict` and `BINARY_MODULO` with a `basestring` have abnormally high frequency/LOC values for the same reason these bytecodes have high frequency numbers as discussed previously.

A correlation list was created based on the outcome of both tables. When

[1] Lines-of-code numbers were calculated using SLOCCount 2.26 [26].

Rank	Bytecode or Method	Argument Types	Frequency
1.	STORE_SUBSCR	dict, -, -	31,786
2.	BINARY_MODULO	basestring, -	8,444
3.	PRINT_ITEM	basestring	3,613
4.	list.append()	list, -	2,587
5.	BINARY_ADD	basestring, basestring	1,148
6.	basestring.join()	basestring	751
7.	BINARY_SUBSCR	dict, -	587
8.	BINARY_SUBSCR	list, -	566
9.	BINARY_LSHIFT	integral, integral	542
10.	BINARY_ADD	integral, integral	534
11.	BINARY_SUBSCR	tuple, -	500
12.	GET_ITER	list	489
13.	GET_ITER	tuple	330
14.	BINARY_MULTIPLY	complex, float	312
15.	BINARY_MULTIPLY	basestring, integral	268
16.	list.extend()	list, list	260
17.	dict.has_key()	dict, -	244
18.	COMPARE_OP	integral, integral	228
19.	INPLACE_ADD	integral, integral	209
20.	INPLACE_ADD	basestring, basestring	205

Table 6.2. **Frequency of Bytecode With Specific Type Arguments**: The "Argument Types" column lists the types of the objects that were expected to be popped off the execution stack and used with the bytecode. "-" represents any type. The first argument is usually what the bytecode is being applied to.

a bytecode/type or method/type was on both lists but at different rankings the preference was arbitrarily placed on the frequency table ranking. From this correlation it was decided that ten bytecodes would be chosen to implement, listed in Table 6.4.

Each bytecode was implemented and the compiler was modified to emit the new bytecodes appropriately. With each new bytecode, a small test was written to measure performance. If performance did not increase for the test, the new

Rank	Bytecode or Method	Argument Types	Frequency/LOC
1.	STORE_SUBSCR	dict, -, -	0.08368
2.	BINARY_MODULO	basestring, -	0.02980
3.	PRINT_ITEM	basestring	0.01036
4.	list.append()	list, -	0.00825
5.	BINARY_ADD	basestring, basestring	0.00239
6.	BINARY_MULITPLY	basestring, integral	0.00202
7.	basestring.join()	basestring, -	0.00166
8.	BINARY_SUBSCR	list, -	0.00161
9.	BINARY_SUBSCR	dict, -	0.00160
10.	BINARY_SUBSCR	tuple, -	0.00154
11.	BINARY_ADD	integral, integral	0.00150
12.	COMPARE_OP	list, list	0.00150
13.	GET_ITER	list	0.00148
14.	BINARY_LSHIFT	integral, integral	0.00085
15.	GET_ITER	tuple	0.00080
16.	INPLACE_ADD	basestring, basestring	0.00079
17.	BINARY_MULTIPLY	complex, float	0.00069
18.	dict.has_key()	dict, -	0.00061
19.	list.extend()	list, list	0.00057
20.	COMPARE_OP	integral, integral	0.00057

Table 6.3. **Frequency/LOC of Bytecode With Specific Type Arguments**: The "Argument Types" column lists the types of the objects that were expected to be popped off the execution stack and used with the bytecode. "-" represents any type. The first argument is usually what the bytecode is being applied to.

bytecode was removed and another bytecode was considered. In some instances it was discovered that bytecodes were already optimized for common types, such as `BINARY_ADD` for arguments of integral type. Others just did not improve performance. The bytecode implementation itself was faster by inspection for the type-specific version, removing at least a couple comparisons and branches and sometimes even a function call in the C implementation. Yet performance still lagged behind the type-agnostic version of the bytecode in some instances. The

loss in performance even without a type-specific optimization was most likely due to cache pressure from Python's main eval loop becoming larger due to the new bytecodes and thus pushing out of the code cache the implementation of the bytecode in the eval loop.

To gain a performance improvement each new bytecode inlines the code needed to perform their actions. Because most of Python's bytecodes are type-agnostic, each bytecode must probe the object passed to it to discover what facilities are provided by the object for implementing the requested action. Since some actions can be implemented in multiple ways, several probings may take place before the final function to be used is discovered.

Consider `STORE_SUBSCR` acting upon a `dict` as an example of a new bytecode that provides a type-specific version of another bytecode. After popping the needed objects off the interpreter stack, a call to the function `PyObject_SetItem()` is made. The function, after checking its arguments, checks if the object implements the mapping protocol and the proper subscription assignment function and calls that function if it exists, else it checks if the sequence protocol is implemented with the needed functions and makes that function call. All of this requires multiple function calls and various comparisons and struct dereferences in the C code. In the `DICT_STORE` implementation, though, a direct call is made to `PyDict_SetItem()` which is the function that is eventually discovered and used by `STORE_SUBSCR`.

For the new bytecodes that replace method calls, the savings are even more dramatic. Method calls typically involve to bytecodes; `LOAD_ATTR` and `CALL_FUNCTION`. The `LOAD_ATTR` bytecode retrieves the method from the attribute it is stored in and pushes it on to the stack. `CALL_FUNCTION` subsequently pops the method object from the interpreter stack and calls it with its arguments. The new byte-

codes that implement a method call shortcut this dual step process and make it a single step process by skipping the creation of the method object to push on to the interpreter stack and the heavy overhead of calling a Python function by executing a C function call directly.

New Bytecode	Bytecode Replaced	Argument Types	Performance Increase
DICT_STORE	STORE_SUBSCR	dict, -	3.2%
STR_FORMAT	BINARY_MODULO	basestring, -	7.8%
LIST_APPEND	list.append()	list, -	28.1%
STR_CONCAT	BINARY_ADD	basestring, basestring	7.7%
STR_MULT	BINARY_MULTIPLY	basestring, integral	8.5%
STR_JOIN	BINARY_ADD	basestring, basestring	16.9%
INT_LSHIFT	BINARY_LSHIFT	integral, integral	13.5%
LIST_CMP	COMPARE_OP	list, list	8.2%
DICT_GETITEM	BINARY_SUBSCR	dict, -	5.8%
DICT_HAS_KEY	dict.has_key()	dict, -	33.8%

Table 6.4. New Type-Specific Bytecodes

Chapter 7

Experiments & Results

Two applications and two actual benchmarks were chosen, listed in Table 7.1, to measure whether the new type-specific bytecodes led to the desired 5% performance increase. All but one (Pyrex) had not been used in the statistics gathering for choosing the new type-specific bytecodes. This was done to ensure that results from choosing the new bytecodes would not pad the results of the benchmarks.

Benchmark	Version	Reference
SpamBayes	1.0rc2	[16]
Pyrex	0.9.3	[8]
PyBench	1.0	[13]
Parrotbench	1.0.4	[21]

Table 7.1. Benchmarks

Before all benchmarks were run, all Python code, including Python's standard library, were explicitly compiled with the `-O` optimization option for both the baseline installation of Python 2.3.4 and the modified version with type inference. Python's `-OO` option was not used because type annotations require the documentation strings to be present; `-OO` strips docstrings from the compiled

code.

The hardware used for benchmarking was an Apple PowerBook G4 1.5 GHz laptop with 786 MB RAM. The operating system was OS X 10.3.7 with all startup applications exited (by holding down the shift key during startup).

Because of the sizable time cost caused by having to modify all docstrings by hand, only two benchmarks were used to measure the benefit of type annotations. Both Parrotbench and Pyrex were modified with type annotations where a known benefit would result and the types passed to the method were known to be consistent. Each benchmark was chosen because they have the fewest LOC and were thus easiest to modify for the initial analysis.

7.1 SpamBayes

7.1.1 About the Benchmark

SpamBayes is a Bayesian filtering program for spam emails [16]. It is written entirely in Python and represents a real-world application. The use of SpamBayes as a benchmark consisted of training it on what was considered good mail (known as "ham") from bad email (known as "spam"). The ham was chosen from six separate months of the python-dev email archives (March 2002, April 2002, February 2003, October 2003, September 2004, and October 2004) totaling 19,474 emails. The spam for the training was from one of the author's personal collections of spam for a day, totaling 34 emails.

Execution consisted of creating a pickled corpus trained on the emails. The command-line arguments to the SpamBayes script `sb_mboxtrain.pyo` were `-p`

`$PICKLEFILE` (with `$PICKLEFILE` representing the name of the pickle file to write to) to create the corpus and `-f` to force a new corpus to be created even if a new one already existed.

Timing was done using the `time` utility. The real time value was recorded.

7.1.2 Results

SpamBayes actually performed worse with the new bytecodes, with performance decreasing by 2.01%. Table 7.2 lists the timings of the test runs.

Type	Run 1	Run 2	Run 3	Run 4	Run 5	Average
Baseline	144.342	145.901	144.842	146.058	145.609	145.35
Type Inference	148.111	148.439	148.873	148.347	148.262	148.41

Table 7.2. SpamBayes Benchmark Results: Timings listed in seconds.

This drop in performance most likely stems from cache performance. If the implementation in the eval loop of type-specific bytecodes must compete for cache space with type-agnostic bytecode implementations, the type-agnostic versions will win due to the lower frequency of use of the type-specific bytecodes.

7.2 PyBench

7.2.1 About the Benchmark

PyBench is a benchmark suite for Python created by eGenix [13]. It performs a wide variety of tests of the Python interpreter's performance on various tasks – from built-in type creation to branching. PyBench automatically performs ten

rounds of experiments and then averages the results to report a final result. The final result reported by PyBench was recorded.

7.2.2 Results

As with SpamBayes, performance actually decreased with the new bytecodes. A quick glance at PyBench suggests that the reason for the performance decrease is the same as that for SpamBayes. But a closer look at the actual results listed in Table 7.3 shows that the base version of Python had one phenomenal run that skewed the average in its favor. Subsequent informal runs of PyBench show that this one phenomenal run is atypical.

Type	Run 1	Run 2	Run 3	Average
Baseline	6719	6712	6603	6678.00
Type Inference	6693	6680	6699	6690.66

Table 7.3. PyBench Benchmark Results: Timings listed in milliseconds.

Obviously a 0.2% decrease in performance is not good. But if you remove the best run from both groups in order to eliminate the anomalous run for the baseline the results change in favor of using the new bytecodes, albeit by a very small margin.

Unfortunately, the performance increase of 0.3% is no where near the 5% aimed for. Because the benchmark covers such a wide swath of situations the possible use of the type-specific bytecodes is not necessarily great.

7.3 Pyrex

7.3.1 About the Benchmark

Pyrex is a modification of the Python programming language to help facilitate creating extension modules that are written in Python but are translated into C [8]. The Pyrex files used for testing were two demo files included with Pyrex, `spam.pyx` and `cheese.pyx`. Because both files are rather small, they were compiled sequentially 100 times per run.

For timing, the `timeit` module from Python's standard library was used [22]. This module was designed to help facilitate the testing of Python code in as accurate a way as possible based on the operating system being used. For testing, the command-line arguments to `timeit` were `-r5` to make the sample come from the top five runs (10 total runs were used as the pool to pull from for the five best runs). The specific Python command passed to `timeit` to execute Pyrex was "`-s 'from Pyrex.Compiler.Main import compile' 'for x in xrange(100): compile("Pyrex/spam.pyx"); compile("Pyrex/cheese.pyx")'`".

7.3.2 Results

Pyrex showed the greatest performance increase with 1%. With the addition of type annotations, the performance improvement rises to 1.6% as shown by the results listed in Table 7.4.

The performance increase can be attributed to the use of string interpolation used extensively throughout Pyrex. The new bytecode `STR_FORMAT` is used in those situations.

Type	Average
Baseline	3.89
Type Inference	3.85
Type Inference, Type Annotations	3.83

Table 7.4. **Pyrex Benchmark Results:Timings in seconds. Average of the best five out of ten runs.**

As for the type annotations, out of approximately 1,165 methods a total of 15 had type annotations added for their arguments. In a single run, a total of 596, 366 method calls are made, of which 88,200 are to type annotated methods. Discussion of the results from type annotations is saved for Chapter 8.

In terms of why so few methods were annotated, it seems that object-oriented programming works against this kind of type annotation. While adding type annotations it became apparent that Pyrex's OOP design led to most data being stored as attributes on the instance. This meant most data was accessed through attribute access rather than being passed around as arguments through method calls. The values of the attributes never had a chance to be taken into consideration by the type inference phase because of the restrictions on inferable types.

7.4 Parrotbench

7.4.1 About the Benchmark

Parrotbench was designed for a competition between the Python development team and the Parrot virtual machine team (the VM to be used for Perl 6) to see

whose interpreter could run Python code faster at OSCON 2004 [21]. Written by Guido van Rossum, the creator of Python, the benchmark uses many advanced features of Python and is considered a compliance test as much as a performance benchmark.

To make sure the benchmark was stressed enough, the number of complete runs through the benchmark in a single execution was increased from 2 to 20. This is done by increasing the number of iterations through all parts of the benchmark in the `b.py` file.

Timing was done using the `time` utility as used by Parrotbench's `Makefile`. The only modification to the `Makefile` was to make sure it used the proper Python interpreter for testing. Otherwise experimentation was done by executing the command `make` and the real time was recorded from the output of `time`.

7.4.2 Results

Like Pyrex, Parrotbench saw a performance increase from both the new bytecodes and the addition of type annotations; 0.7% and 0.8%, respectively as shown by the results listed in Table 7.5.

Type	Run 1	Run 2	Run 3	Run 4	Run 5	Average
Baseline	224.53	224.227	224.195	224.214	224.399	224.313
Type Inference	222.756	222.614	222.796	222.834	222.454	222.691
Type Inference, Type Annotations	222.422	222.351	222.394	222.523	222.611	222.460

Table 7.5. Parrotbench Benchmark Results:Timings in seconds.

As with the other tests, the results support the idea that the algorithm and its implementation just cannot infer enough type information to be of great use.

Only three of the 160 methods in Parrotbench benefited from type annotations. Out of 394,871 method calls in a single run, 113,667 of those calls are to type annotated methods. Discussion of these results are deferred to Chapter 8.

Chapter 8

Conclusion

Introducing over 3,000 lines of new C code to Python's compiler to get, at best, a 1% improvement is in no way justified. The level of added complexity that would be introduced into the compilation step would definitely need to lead to a noticeable performance improvement, the 5% that was the goal, to justify the cost of code maintenance.

As noticed while adding type annotations, a large amount of data is stored within objects and is not passed around between methods. Typically local variables were used as temporary variables. Within an object-oriented design, temporary values can be stored in the attribute that the value will end up in, thus removing the need for a local variable. This appears to be the situation in the benchmarking code.

Object-oriented programming also affects the effectiveness of type inference. As discussed in Chapter 7.3.2, under the imposed semantic limitations, inferring the types of object attributes is not possible. With heavy use of object attributes encouraged as good object-oriented design, this poses a severe limitation that

cannot be overcome without a change in the language or restrictions of the algorithm.

There also seems to be no consistent benefit from type annotations. Pyrex was able to gain a 0.6% performance increase from type annotating 1.3% of its methods which are subsequently called 14.8% of the time. Compare this to Parrotbench, which had a 0.1% performance increase from adding type annotations to 1.9% of its methods which are called 28.8% of the time. Pyrex gets a six-fold performance increase over Parrotbench's with fewer annotated methods and fewer calls to those methods. This suggests that type annotations can increase performance but as to how effective they are per method and overall heavily depends on how the performance benefit manifests itself within the method, as expected.

There is the possibility that fewer new bytecodes, in order to shrink the size of the eval loop in Python's interpreter to improve cache performance, might actually improve performance. That would require very extensive testing on multiple platforms to make sure the benefit was not in any way architecture-dependent. Typically this is fairly difficult to get right and could easily be lost when the next "great" CPU is released. This does not seem like a viable option nor necessarily the reason for the poor performance overall.

In the end, it seems that Python, as a language, is not geared towards type inference. Its flexibility, considered a great strength, interferes with how extensive type inference can be performed. Without a significant change to the language, type inference is not worth the hassle of implementation.

Chapter 9

Future Work

There are multiple possibilities for future work based on the results of this thesis. For one, if one were to remove the restriction of not changing the semantics of the compiler, improvements might be possible. One could add a check between compiled code and its corresponding source code at run-time. If there is no time delta putting the source code as a more recently touched file then one can trust that compiled code to be current. If there is no source code and only compiled bytecodes, then an assumption that the bytecodes are trustworthy can be made. With that level of trust, and a promise from the developer that no other code will be introduced (this can be implicit through some execution flag such as `-OO`) then there is nothing technically stopping full type inference of Python code.

Analysis of whether introducing a keyword to signify when something in the built-in namespace is allowed to be shadowed might warrant investigation. Prevention of unexpected shadowing of the built-in namespace from external code to the module would be the purpose of this addition to the language. This would allow the compiler to know when built-ins would have to be considered unknown or could be trusted to be what they are as defined by default. Return types on

built-in functions could then be used in type inference and in the emission of bytecodes. It would also allow for bytecodes that represented built-in functions directly. This obviously does not deal with the global namespace in any way, but it would expand what type information is available.

The possibility of doing a more thorough implementation of the algorithm is always possible. If that is done, though, it should wait until the AST branch for Python is completed. This branch in CVS for Python will change the compiler to follow the much more traditional "parse tree → AST → CFG → bytecode" steps taken by most compilers. This should lead to a simpler implementation and thus allow a much more accurate implementation of the algorithm.

In terms of algorithm improvement, a few possibilities exist. There might be a way to take `break` and `continue` statements in loops in a more careful manner to somehow allow using a flow-sensitive type inference of the loop body in some way. One could also keep both a flow-sensitive and flow-insensitive type inference going at all times to allow for the inference of the `else` block of a `try/finally` statement with the flow-sensitive version of the `try` block.

Finally, future work could focus on gathering type information at run-time. Using type feedback, in which one keeps track of the types used in expressions at run-time, it would be possible to detect at run-time consistent type usage [10]. With this information various optimizations would be possible thanks to hotspot recompilation where the type information is helpful and seemingly stable enough to warrant the cost of recompilation and skipping type inference's compile-time restriction. To clarify, this is not what Psyco does; Psyco uses JIT compilation and bases its decisions on optimizations made at compile-time instead of run-time [18].

One could even remove the entire need for run-time recompilation by outputting run-time type information from type feedback. This data could then be fed into the compiler with a second attempt at compilation to optimize for type information. The performance penalty of having to recompile at run-time would thus be removed, although it would require a profiling run of the code which in no way guarantees the type information gathered will be typical of most runs of the code.

Bibliography

[1] O. Agesen. Constraint-based type inference and parametric polymorphism. In First International Static Analysis Symposium, pages 78–100, 1994.

[2] O. Agesen. The cartesian product algorithm. In ECOOP, August 1995.

[3] K. Altis. PythonCard. http://pythoncard.sourceforge.net/.

[4] L. Cardelli. Basic polymorphic typechecking. Science of Computer Programming, 8(2):147–172, 1987.

[5] C. Chambers and D. Ungar. Iterative type analysis and extended message splitting: Optimizing dynamically-typed object-oriented programs. In LISP AND SYMBOLIC COMPUTATION: An International Journal. Kluwer Academic Publishers, 1991.

[6] B. Cohen. Bittorrent. http://www.bittorrent.com/.

[7] Enthought. SciPy. http://www.scipy.org/.

[8] G. Ewing. Pyrex. http://www.cosc.canterbury.ac.nz/ greg/python/Pyrex/.

[9] B. Heeren, J. Hage, and D. Swiestra. Generalizing Hindley-Milner type inference algorithms. Technical Report UU-CS-2002-031, Institute of Information and Computing Sciences, Urecth University, 2002-07-08.

[10] U. Hölzle and D. Ungar. Reconciling responsiveness with performance in pure object-oriented languages. In **Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '94)**, pages 229–243. ACM Press, October 1994.

[11] M. P. Jones. Functional programming with overloading and higher-order polymorphism. In **Advanced Functional Programming**, pages 97–136, 1995.

[12] T. M. Labs. Twisted. http://twistedmatrix.com/products/twisted.

[13] M.-A. Lemburg. PyBench. http://www.egenix.com/files/python/pybench-1.0.zip.

[14] A. Limi, A. Runyan, and V. Andersen. Plone. http://plone.org/.

[15] R. Milner, M. Tofte, R. Harper, and D. MacQueen. **The Definition of Standard ML (Revised)**. The MIT Press, 1997.

[16] T. Peters. SpamBayes. http://spambayes.sourceforge.net/.

[17] PythonWare. Python imaging library. http://www.pythonware.com/products/pil/.

[18] A. Rigo. Psyco, the Python specializing compiler, December 2001.

[19] M. Salib. Faster than C: Static type inference with Starkiller. Master's thesis, MIT, 2004.

[20] G. van Rossum. Optional static typing – stop the flames! http://www.artima.com/weblogs/viewpost.jsp?thread=87182.

[21] G. van Rossum. Parrotbench. ftp://ftp.python.org/pub/python/parrotbench/parrotbench.tgz.

[22] G. van Rossum. Python library reference. http://www.python.org/doc/2.3.4/lib/lib.html.

[23] G. van Rossum. Python language reference. http://docs.python.org/ref/ref.html, November 2004.

[24] K. Walker and R. E. Griswold. Type inference in the Icon programming language, March 1996.

[25] B. Warsaw. Mailman. http://www.gnu.org/software/mailman/.

[26] D. A. Wheeler. SLOCCount. http://www.dwheeler.com/sloccount/.