Starkiller: A Static Type Inferencer and Compiler for Python

by

Michael Salib

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2004

Author .....
Department of Electrical Engineering and Computer Science
May 20, 2004

Certified by .....
Jonathan Bachrach
Research Scientist
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Starkiller: A Static Type Inferencer and Compiler for Python

by
Michael Salib

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 2004, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Starkiller is a type inferencer and compiler for the dynamic language Python designed to generate fast native code. It analyzes Python source programs and converts them into equivalent C++ programs. Starkiller's type inference algorithm is based on the Cartesian Product Algorithm but has been significantly modified to support a radically different language. It includes an External Type Description Language that enables extension authors to document how their foreign code extensions interact with Python. This enables Starkiller to analyze Python code that interacts with foreign code written in C, C++, or Fortran. The type inference algorithm also handles data polymorphism in addition to parametric polymorphism, thus improving precision. Starkiller supports the entire Python language except for dynamic code insertion features such as `eval` and dynamic module loading. While the system is not yet complete, early numeric benchmarks show that Starkiller compiled code performs almost as well as hand made C code and substantially better than alternative Python compilers.

Thesis Supervisor: Jonathan Bachrach
Title: Research Scientist

Acknowledgments

This work is a product of my time at the Dynamic Languages group, and to them I owe an enormous debt. Jonathan Bachrach supervised this thesis and provided invaluable advice throughout. As a thesis adviser, he combined an encyclopedic knowledge of dynamic languages and compilers together with the patience of Job. Greg Sullivan guided this work at a very early stage and was the one who suggested that I look into Ole Agesen's work. Howie Shrobe provided encouragement and advice, and somehow always managed to find the funding needed to support this work. In different ways, all three have served as role models. If I know anything about juggling chainsaws, it is because I learned by watching them.

I would also like to thank Ole Agesen since much of the work here is based on his research at Stanford.

I cannot thank enough my parents, who never stopped fighting for me, no matter how hopeless the situation. I'd also like to thank my brother and my grandparents for teaching me how decent human beings live. Siessa Kaldas provided food and shelter at a critical time in the writing of this thesis. If it were not for the hard work of Anne Hunter and Kristine Girard, I never would have made it this far. Finally, I must thank Benazeer, who has become a guiding light in my life.

Looking back, I can see that I have been blessed beyond measure; more than anything, I am grateful for the hope that has sustained me these long years, given to me by my friends and family.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Information technology has brought marvelous advances to our society. At the same time, it has extracted a terrible cost, and that cost is growing. From the software developer's perspective, software costs too much and takes too long to build. Indeed, empirical evidence suggests that the majority of large software projects are completed significantly over budget and beyond schedule [10]. The situation is even worse from the software user's perspective: she sees a product that is fragile, defective, and dangerous. Calling most commercial software "dangerous" may seem like an exaggeration until one asks the opinions of the millions of people forced to spend billions of dollars cleaning up after an unending series of viruses and worms infecting various pieces of Microsoft software [42, 43, 44, 45].

1.1.1 The Coming Plague

Project Oxygen espouses a vision in which computing becomes as ubiquitous as the very air we breathe. But without radical changes in how we design and build software systems, the realization of that vision will bring about a dystopia where software failures and malicious attacks thoroughly poison all aspects of our lives. As bad as the current situation may be, it will become a lot worse. Computers are widespread in our society but as trends in microelectronics further reduce the cost of computational elements, they will become pervasive. The value of this universal computation is greatly compounded by universal information access. Thus, the drive to add computational agents anywhere and everywhere is tightly bound to the drive for universal network connectivity.

We are thus entering a world where every system is a distributed system; everything is connected, and all those connections bring with them new failure modes and new attackers. This poses a double problem: not only will we be increasingly dependent on computers in all aspects of our lives, but because those computers will be networked, they will be all the more vulnerable. In fact, Leslie Lamport humorously describes a distributed system as "one in which the failure of a computer you didn't even know existed can render your own computer unusable."

Of course, right now you can still use the postal service should your computer be disabled by the latest Microsoft worm. However, the day may yet come when the postal service requires computer generated addressing for delivery (after all, it is much cheaper than handwriting recognition). Likewise, you can still use the telephone network in the event of a desktop computer failure, but major telephone network operators have already begun integrating their voice and data networks [41] and many large companies have abandoned dedicated voice service entirely in favor of Voice over Internet Protocol. Indeed, the cellular phone is a perfect example of such convergence. Modern cellular phones are essentially portable computers coupled to transceivers. Newer cell phones run Linux and other operating systems designed originally for desktop computers and consequently carry many of the same vulnerabilities [36, 34].

1.1.2 Dodging Bullets

The horrific vision of a helpless population utterly dependent on increasingly vulnerable and fragile technology begs the question: "how do we improve software reliability and security?". As it happens, software reliability and security are intimately related. In fact, the vast majority of remotely exploitable network vulnerabilities are caused by buffer overruns [33]. Computer scientists have known about these problems for the past 40 years and have observed them causing exploited vulnerabilities in the wild for the past 30. This example suggests that the use of low level programming languages is partially responsible for software unreliability and insecurity. Higher level languages (HLLs) eliminate buffer overflows by either perform range checking for each array access or statically verifying that range violations are impossible.

In fact, HLLs can alleviate many of the problems plaguing software development described above. HLL programs are typically far shorter than equivalent lower level language programs. Research indicates that the number of programmer mistakes made as well as the time needed to write a program are proportional to the length of the program. This suggests that programs written in an HLL can be written faster and have fewer bugs than equivalent programs written in lower level languages [30]. Because writing code takes less time, its easier for developers to prototype designs allowing them to get better user feedback [4] and to experiment with alternative designs quickly [6]. Both of these techniques significantly improve the user experience. In addition, HLLs often benefit from the last 30 years of research in software engineering in ways that lower level languages and their immediate descendants rarely do. For example, Python's syntax was designed in part to eliminate the dangling-else problem that often causes serious bugs in C-like languages. The importance of avoiding the dangling-else problem was brought home on January 15, 1990, when most of the national telephone network went offline because one programmer misplaced a single closing brace due to the dangling-else problem [39].

1.1.3 Where is the Red Pill?

All this raises the question of why haven't HLLs succeeded in the marketplace given their clear benefits. There are many reasons for this, but two stand out: performance

and familiarity. Regardless of how excellent a language is, it provides society no benefit if no one uses it. Many HLLs look completely alien to everyday programmers, either because of syntax (drowning in Lisp's sea of parenthesis) or semantics (the total absence of mutable state in Haskell) or both (Smalltalk, just because). For an example of how important familiarity is, consider garbage collection. It took enormous time and energy for Sun Microsystems to convince the larger programming community to accept the use of GC in Java, even though it has been a widely used technique for the last 30 years and solves very real problems.

While some HLLs boast impressive performance, serious problems remain. In particular, Perl and Python, the most popular HLLs, exhibit poor run time performance. Currently, both of these languages suffer from performance problems in their virtual machine implementation unrelated to the languages themselves [40]. Even if we had the perfect virtual machine, however, these languages would still suffer because they are dynamically typed. As Shivers points out in [37], compiler optimization algorithms for dynamically typed languages have lagged far behind their statically typed cousins. The crux of the problem is that dynamic typing makes it impossible to statically determine the control flow graph and without that, many traditional optimization techniques cannot be applied. Agesen explains that the problem is even worse for object oriented languages since dynamic dispatch prevents the application of many optimization techniques for much the same reason [1].

Many features of the Python language impede compilation. For example, in contrast to C and C++ where function and method definitions are declarative and resolved statically, in Python, these definitions are imperative, thus postponing function and method resolution until run-rime. A definition statement in Python simply binds the function's name to a newly created function object. Moreover, because functions are first class objects, nested function definitions require that the function objects corresponding to inner definitions be repeatedly created on each invocation of the outer definition. For example, in the snippet of code shown in Figure 1-1, a new version of the g function will be wastefully created and returned for each invocation of the function f.

```
def f():
    def g():
        return "spam"
    return g
```

Figure 1-1: A function that returns another function

Python's object system provides further barriers to efficient compilation. More specifically, a class' list of base classes as well as its list of methods can change at run time. Because of this dynamic inheritance and method binding, many traditional object oriented programming language optimizations become inapplicable. This is because even when a compiler can statically determine what class an instance belongs to, it cannot safely inline method bodies for that instance since neither the class' set of

methods nor its base classes can be statically determined at runtime from declarative elements in the program source code. Most strangely of all, an instance's class can be changed dynamically as shown in Figure 1-2.

```
class Person:
    pass

class Professor:
    pass

bob = Person

bob.__class__ = Professor
```

Figure 1-2: Pathological program demonstrating dynamic class membership.

The object system provides further obstacles to simple compilation by maintaining strong reflection and introspection properties. For example, instance or class attributes can be added at runtime using names that are generated at runtime by means of the setattr function. In practice, this means it is impossible for any compiler to know the complete set of attribute names associated with certain classes or instances. In a similar manner, Python's inclusion of an eval function (and its related exec function) make it impossible to statically examine all code that will be executed.

1.2 Past Type-Inference Algorithms

Despite the fact that existing languages and compilers fail to provide the level of safety and performance needed to avert the aforementioned problems, the literature on type inference is extensive. Much of this research centers on purely theoretical analysis with little applicability to real world programs. Moreover, much of the practical research centers on improving type inference for advanced languages too unpalatable for the average programmer. Nevertheless, while current literature is unable to provide an immediate solution to the problems described above, it can certainly provide the foundation for such a solution. I review the relevant research presently. Note that the structure of this review is partially based on Ole Agesen's Thesis [1].

1.2.1 Hindley–Milner

Hindley invented a type inference algorithm that was later independently re-invented by Milner; a concise introduction can be found in [7]. The algorithm operates by annotating the nodes of an expression tree with type constraints in a bottom-up fashion. Type constraints consist of either primitive types, type variables, or function types. When processing an expression tree, constant elements are tagged with their corresponding primitive types. Variable references are tagged with the type variable

of the corresponding variable. Compound expressions introduce new constraints on their components to ensure that they fit together properly.

Ill-formed programs generate inconsistencies in the set of deduced constraints that cause type inference to fail. Such inconsistencies indicate that a program may fail to execute correctly. Cartwright and Fagan later extended basic Hindley–Milner with soft typing. Compilers that use soft typing do not reject programs that generate inconsistent types, but rather insert code to detect type errors at run-time in parts of the program that cannot be statically verified.

Hindley–Milner forms the basis of type systems in many experimental languages such as Haskell and the ML family of languages. It is often, but not always, implemented with unification. Unfortunately, it suffers from a number of drawbacks. Principal among them is that it infers the most general type possible. Specifically, type variables are initially unrestricted and are only further constrained to the extent needed to eliminate run-time type errors. This strategy has significant implications for performance, since optimal performance typically demands knowledge of the concrete, or most specific types of each expression. Another problem with Hindley–Milner type inference is that it was designed for functional languages with no notion of state. It has since been extended to support mixed imperative and functional paradigms as well as object oriented semantics, but with varying degrees of success.

1.2.2 Shivers

Shivers began by considering why Lisp compilers failed to optimize as efficiently as Fortran compilers on equivalent programs. He concluded that the discrepancy was due to the inability of Lisp compilers to statically determine a program's global call graph, which in turn precluded the application of many traditional data flow analysis algorithms. Shivers further claimed that the presence of higher order functions greatly impedes construction of static call graphs because they obscure function call targets. As a result, Shivers recommends that closures must be tracked to their corresponding call sites [37]. Much of his resulting work is theoretical and what little practical work there is eliminates the possibility of analyzing programs with imperative semantics.

Nevertheless, Agesen made the connection that the problem of type inference in the presence of higher order functions as described by Shivers is much the same problem seen in object oriented languages with dynamic dispatch [1]. By analogy, Shivers' work suggests that objects must be tracked to sites from which their methods are called.

1.2.3 Palsberg and Schwartzbach

Palsberg and Schwartzbach's basic algorithm for type inference is totally unsuitable for practical implementation, but formed the basis for Agesen's later work and provides an excellent vehicle for understanding why many type inference algorithms fail. The algorithm constructs a constraint network in which the type of each expression in the program is represented as a node. The value of a node consists of a (possibly empty) set of types which that expression might assume during program execution.

Constraints between nodes are imposed by program statements. Initially, all type sets are set to the empty set except for constants and literals. The type sets for these expressions are initialized to a single element containing the type of the constant or literal. Finally, the algorithm propagates types along constraint edges. Constraints are designed to mimic run time data flow and thus impose a subset relation on the two nodes they connect. In other words, if expression x's type set is connected to expression y's type set, whenever a new type is added to x, it will propagate along the constraint and be added to expression y's type set. At this point, expression y will propagate the new type to any nodes it has constraints to. This propagation process continues until there are no unseen types in the network, i.e., when the constraint network has reached equilibrium.

Constraints are added based on two features of the program text: assignment and function calls. For a given assignment of the form $x = y$, the algorithm creates a constraint from the type set for expression y to the type set for expression x. This enforces the superset relation ensuring that the set of types which x can achieve during execution must be a superset of the set of types y can achieve. Function calls are dealt with by building constraints from each actual argument to the corresponding formal argument. For example, given a function call $f(3, 4.0)$ and a function definition for $f(x, y)$, Palsberg and Schwartzbach's algorithm would build a constraint between the type set for the expression 3 and the variable x in the body of f's definition in addition to a constraint between the type set of 4.0 and the variable y in f's body.

Palsberg and Schwartzbach enhanced this algorithm to improve inferencer precision of object oriented code by making several changes. They eliminate inheritance by copying base classes into the bodies of their descendant classes. This ensures that the effects of multiple descendant classes are not commingled in the same code block. In a similar vein, they duplicate class definitions at each constructor call site and duplicate method bodies at each method call site. These techniques improve the algorithm's ability to operate precisely in the presence of data polymorphism and parametric polymorphism respectively.

It is important to realize that these expansions are done once, and not recursively. Later versions of this algorithm repeated the expansion a fixed number of times. While solving a number of problems, this basic algorithm has a number of serious flaws. In the worst case, successive expansions squared the size of the program to be analyzed. Yet without them the algorithm proved to be too imprecise for practical use. Moreover, much of the analysis performed by the algorithm was unnecessary since code was expanded regardless of whether precision required expansion or not.

1.2.4 Agesen

Agesen based his work on that of Palsberg and Schwartzbach. He introduced a new type inference algorithm designed to remedy the worst faults of theirs and called it the Cartesian Product Algorithm (CPA). In contrast to the basic algorithm's fixed expansions, CPA attempts to adapt in a flexible manner to the amount of parametric polymorphism present in a program. It strives to do no more work than is strictly necessary to achieve maximal precision.

The specific problem that CPA tries to remedy is the case where a function is called in with different argument types. The basic algorithm described above will conflate argument types from different calls thus reducing precision. By performing repeated expansions, this problem can be lessened, but not eliminated. CPA solves this problem by building multiple copies of each function body called templates. Each template corresponds to an argument list containing only monomorphic types, that is, type sets containing exactly one type. At each function call site, CPA takes the cartesian product of the list of argument type sets, creating a set of argument lists containing only monomorphic types. Each of these argument lists is then connected to its corresponding template body; the return values of all the templates generated for a particular call are all connected to the result of the function call.

The key to CPA's operation is that template bodies are reused. This ensures that CPA never analyzes the same function more than once for each possible set of argument types. CPA is efficient in that it never repeats an analysis. For example, if a new type is added to a function call argument, CPA will take the cartesian product of the resulting argument type set list, but most of the templates needed for that call will have already been analyzed and are already connected. At that point, CPA will only build templates for the monomorphic type set argument lists that have not been seen before. Furthermore, since each template has monomorphic arguments, no precision is lost due to parametric polymorphism: calls to the same function with different types are not commingled but remain isolated in different template bodies. Their only interaction occurs when their results are integrated into the function call return value.

The core idea behind CPA is monotonicity. In other words, CPA always maintains correct, but potentially incomplete type information for a program. As new information is deduced, it propagates through the constraint network. Consequently, CPA is far more efficient than the basic algorithm described above and is also far more precise. However, it does have several problems, especially when applied to a language like Python. While it handles parametric polymorphism well, it performs imprecisely in the presence of data polymorphism. It also has no way of dealing with foreign code linked into the program from external sources but written in a different language from the main program.

The last major problem Agesen identified has to do with recursive closures. Essentially, type inference of a function that recursively calls itself may never terminate. This happens when CPA builds a template for a recursive function but in so doing must build a template for the recursive call inside the original recursive function. If the recursive function call in the template was connected back to itself, analysis would terminate and there would be no problem. But, the two templates may differ because their lexical environments differ. This can easily happen when one of the arguments is a closure since on each successive recursive invocation, the closure will be different. Agesen introduced heuristics into his CPA implementation to try and check for unbound recursive analysis, but this solution is somewhat fragile and very ad-hoc.

1.3 Python Optimization Tools

Python's longstanding performance problems have not been ignored by the community; far from it in fact. There have been several significant efforts over the years directed at improving Python's run time performance through compilation. Nevertheless, none of these efforts has proven to be a sustainable, general purpose, easy to use solution to Python's performance problems. I survey the more significant of these efforts below. Two of these tools rely entirely on static compilation while the other two utilize dynamic compilation. It is telling that both of the static compilation projects have died out while the dynamic compilation projects remain in active use and see continued development. In addition, as will be shown in Chapter 4, tools that perform dynamic compilation outperform those that rely on static compilation. Part of this discrepancy may be explained by the fact that the dynamic compilation tools generate specialized versions of compiled code based on the types observed at runtime while the static compilation tools essentially generate slower generic code since they lack both the ability to observe runtime types as well as the ability to infer types statically. Consequently, the availability and relative performance of optimization tools for Python highlights the vital importance of static type inference for supporting static compilation.

1.3.1 Python2C

Python2C is a straightforward Python to C compiler written by Bill Tut and Greg Stein. Given Python module source file, it generates equivalent code in C. Since it performs no type inference and little optimization, the code it generates consists almost entirely of calls into Python's C API. Optimizations that are performed include directly calling by name functions defined in the top level (rather than calling them indirectly through a module dictionary), interning of integer and string constants, optimized for–loop representations when the loop bounds are constant integers, and inlining integer specific code for binary operations. Note that since Python2C does not perform type inference, the inlining just described includes type checks on the operands; should either operand fail to be an integer, the slower generic binary operation is called.

By effectively "unrolling" the virtual machine, Python2C eliminates much of the overhead associated with byte code processing and dispatch. Those benefits come at the cost of a large increase in code size, which in turn can incur a significan performance penalty by increasing memory pressure and overflowing the instruction cache. As might be expected, performance of the resulting code is mixed. Anecdotal evidence indicates that programs compiled with Python2C see only a small performance boost on average, about 20%. Some programs experience performance regressions when compiled with Python2C. Development of Python2C terminated in 2000 and has not resumed since. Ironically, during the interim period, the performance of the CPython virtual machine has improved by an equivalent amount due simply to incremental enhancements. More specifically, Python 2.3.3 has a 15% higher score on the PyStone benchmark than Python 1.5.2 has.

1.3.2 211

211 is a program that compiles Python bytecode into equivalent C code and then integrates the generated C code into the Python virtual machine [2]. It splits sequences of bytecodes into Extended Basic Blocks (EBB) and, for each EBB, generates a fragment of C code equivalent to concatenating the C code executed by the VM for each bytecode in that EBB. The C code for proccessing each EBB is associated with a new bytecode and is implanted directly into the VM's dispatch loop. In other words, long sequences of bytecodes are replaced by a single bytecode, custom generated for the program being analyzed. The code to handle this new bytecode is spliced into the VM, necessitating a recompilation of the VM as well as translation of the original bytecode file into one that utilizes the new bytecodes.

211 goes a step further however: it runs a peephole optimizer over the new bytecode handlers it generates. This optimization pass removes performance degrading artifcats that result from concatenating individual bytecode handlers together. For example, it eliminates adjacent push and pop instructions. In addition, the extra optimization pass presents new opportunities for optimization to the platform's C compiler.

As in the case of Python2C, the results of using 211 are mixed at best. Tests by Aycock indicate a 6% performance improvement over the standard Python VM on the PyStone benchmark. The disappointing overall performance improvement may can be attributed in part by the increased code size: 211's extra bytecode handlers made the VM dispatch loop file 5 times larger than it was before. It should be noted that on some micro-benchmarks, 211 performed substantially better than CPython. It particularly excelled when compiling for–loops and conditionals.

1.3.3 Weave

Weave [24] is a collection of tools designed to improve Python's performance in scientific computing. It includes several components, but only one is of interest here, the Blitz package. Blitz exposes a Python function that compiles a string of Python code into C++, compiles the resulting C++ code with the platform C++ compiler, dynamically loads the generated module, and executes the code. Unfortunately, the C++ compilation stage takes a great deal of time due to the complexities of the underlying C++ libraries. Consequently, Blitz caches compiled code aggressively. A short example of Blitz in action taken from [24] is shown in Figure 1-3.

Blitz does not perform type inference per se, but does perform runtime specialization. Essentially, it checks the types of all variables referenced in the given statement at runtime before compilation and compiles a version of the code specialized for those operand types. Blitz improves upon Numeric Python's performance by eliminating temporary variables and performing aggressive loop fusion. For the scientific computing problems for which Blitz was designed, such optimizations provide significant performance benefits. However, these benefits come at the cost of generality. Blitz only handles expressions involving Python scalars and Numeric Python array objects. The only type of statement that Blitz can compile is assignment to an existing

```
from scipy import *
import weave
a = ones((512,512), Float64)
b = ones((512,512), Float64)

weave.blitz(
   """a[1:-1,1:-1] = (b[1:-1,1:-1] + b[2:,1:-1]
                    + b[:-2,1:-1]+ b[1:-1,2:]
                    + b[1:-1,:-2]) / 5.""")
```

Figure 1-3: Using Blitz to make a two-dimensional average faster

variable.

Yet another problem Blitz faces is that it implements semantics that differ subtly from the ones that Numeric Python implements. For example, in cases where a variable's value is being used to update itself, Numeric mandates the creation of temporary objects to ensure that the entire right hand side of the assignment has completed before the left hand side is modified whereas Blitz will simply modify the variable being assigned to in-place.

1.3.4 Psyco

Psyco [31, 28] is the most recent and most promising of the compilation technologies surveyed here. It is a just-in-time specializing compiler. More specifically, Psyco replaces the main interpreter loop with an internal function that examines bytecodes to be executed and generates specialized code to execute in their steed. This generated code is raw i386 machine code that has been specialized for the types of operands and variables referenced. To reduce overhead, Psyco caches generated code aggressively, which can lead to memory exhaustion when used indiscriminately in large and complex applications.

The primary benefit of Psyco's specialization algorithm is the ability to integrate constant values and types whose invariance is not statically determinable. This yields substantial performance improvements because when Psyco generates code specialized for particular objects, it need not construct the complete object representation. Since the synthesized code "knows" precisely what the types are, it can use a much more compact and immediate representation. For example, integer objects are at least three machine words long and must be heap allocated (albeit with special purpose allocator). All access to integer objects occurs via pointer. Psyco can adapt a much simpler representation, namely a single machine register that contains the integer value. In the event that types change and the current specialized code is no longer applicable, execution falls back to a slower, generic version of the code. The reversion to generic code requires that "virtualized" objects be converted into real objects based on their compact representation so normal Python code can interact with them.

1.4 Starkiller in Historical Context

Starkiller's type inference algorithm is based loosely on Agesen's Cartesian Product Algorithm [1], but with significant changes needed to support an entirely different language (Python versus Self) and to rectify serious problems in the algorithm. CPA has three main problems that Starkiller solves: recursive closures, data polymorphism, and interactions with foreign code.

Starkiller dispatches with Agesen's clumsy method for dealing with recursive closures. Instead, function bodies maintain a set of "silent arguments" associated with them corresponding to all of the out of scope variables those bodies close over. Starkiller treats these silent arguments just like it treats a function's normal arguments; they participate in template formation and selection using the cartesian product. This approach eliminate problems Agesen encountered with indirect recursive customization.

Data polymorphism proves to be a more serious challenge, but one that must be tackled to precisely analyze common Python programs. Starkiller deals with data polymorphism by creating unique polymorphic instance types for each constructor call site. These types contain a type set for each instance attribute. When a method is called on an instance type, the method is treated like a function which takes the instance object as its first argument. However, instead of passing in the instance type as the type of the first argument, the instance type takes the cartesian product of its attributes and generate a list of monomorphic instance state types. These instance states have a link back to their parent instance type so that they proxy all state changes back to the original instance type. As a result, instance state types are completely immutable; any changes made to them are proxied back to their polymorphic parent type who then generates new instance state types. These new instance state types propagate from the constructor call site throughout the rest of the program via data flow, eventually reaching everywhere their earlier forebears reached. Method templates are shared between all users of a particular instance, but not across all instances of a class. This is because instance state types maintain a link back to their respective parents; if method templates were shared, a method induced attribute type change would effect only one instance type rather than all of them.

Finally, Starkiller provides a mechanism for dealing with code that is used by Python programs but that is not written in Python and is not accessible for analysis. Since much of the functionality in the Python library depends heavily on external modules written in C, this issue is vitally important for practical deployments. Consequently, we provide a way for extension module authors to describe how their extensions interact with the type system. For example, the Python function `str` takes an object and, if it returns, always returns a string. Unfortunately, while many external modules have similarly simple type behaviors, there are some that exhibit far more complex behavior. For example, the function map takes a list of elements and a function but returns a list of elements which have a type corresponding to the return type of the given function when passed an element of the input list as an argument. Furthermore, some external functions modify the state of Python objects passed to them. Since the Python/C API provides extension authors with complete

access to the Python runtime, an extension can do anything to objects in the system that normal Python code can do.

This thesis describes the design and implementation of Starkiller with a particular focus on its type inference algorithm. Chapter 1 motivates the subject and reviews past work. Chapter 2 describes Starkiller's type inference algorithm in depth. Chapter 3 describes Starkiller's compiler. Chapter 4 presents an analysis of how Starkiller fares compared to similar systems. Finally, Chapter 5 reviews contributions made in this thesis and makes suggestions for future work.

Chapter 2

Type-Inferencer Design

Starkiller's type inference algorithm is designed to take as input a Python source program and generate as output a declaration of what type each expression in that source program will achieve at runtime. There are two unexpected features of Starkiller's output that bear mentioning. The first is that each function and method in the declaration will be repeated such that each copy is specialized for a particular sequence of monomorphic argument types. The second feature is that a single class may produce more than one class instance type after construction. These features are artifacts of the manner in which Starkiller deals with parametric and data polymorphism respectively. Nevertheless, they are useful artifacts since they naturally suggest particularly efficient compilation strategies.

From a type inference perspective, Python is a large and complex language. In contrast to other languages that rely heavily on type inference for performance, such as Eiffel, Haskell, or the many variants of ML, Python was not designed with any thought as to how the language semantics would hinder or help type inference. Instead, Python's semantics evolved over several years in response to feedback from a community of skilled practitioners. Thus, while languages like Haskell suffer occasional design flaws that had to be imposed specifically to make type inference easier, Python makes no such compromises in its design, which only makes Starkiller's job that much harder. One example of type inference dictating language development is how Haskell forbids the polymorphic use of functions passed as arguments because the Hindley-Milner type inference algorithm that Haskell relies upon cannot handle such nonlocal usage. This limitation stems directly from Hindley-Milner's focus on incremental inference rather than whole program analysis.

While Starkiller's TI algorithm can analyze a large subset of the Python language, there are some language constructs that it cannot handle. Programs that use these constructs are rejected by the inferencer. Most unhandled constructs have a known algorithm for type inference but have not been implemented due to lack of time. There are, however, several language features that remain unimplemented because we do not know how to perform type inference in their presence. These features introduce new code into the system at runtime and thus necessarily render static type inference impossible. These features are dynamic module loading, and the functions `eval` and `exec`. In practice, most programs that would benefit from static type inference and

compilation do not make use of these features, so their exclusion from Starkiller is of little concern. Nevertheless, in Section 5.2, we discuss techniques that may one day allow the integration of these features into Starkiller.

Like Agesen [1], we necessarily take an operational approach in describing Starkiller's TI algorithm. We thus forego mathematically intensive descriptions of the denotational semantics in favor of simpler step-by-step descriptions of what the TI algorithm does.

To ease exposition, we begin by describing Starkiller's TI algorithm for the most basic language features: assignments, function definitions and calls, and the object system. We then examine the TI algorithm as it relates to more advanced but peripheral features of Python, such as generators and exception handling. Having completed a thorough description of how Starkiller's TI algorithm works for pure Python code, we then turn our attention to Starkiller's foreign code support and review how Starkiller integrates the analysis of foreign code extensions into its analytic framework. We explore the subject in some detail by examining how Starkiller handles the built-in functions and types. Finally, we review the most serious of Starkiller's problems and limitations.

2.1 Core TI Algorithm

The core idea behind Starkiller's type inference algorithm is to find concrete types in the program source and then propagate them through a dataflow network that models the dynamics of runtime data transfer. Concrete types are those types that are actually created at runtime, as opposed to an abstract type that is either never created (such as an abstract base class) at runtime or that is a model for any number of types (as in Hindley-Milner type inference). Because Starkiller deals exclusively in concrete types, it must have access to the entire input program source at once. In other words, it must perform whole program analysis rather than incremental one module at a time analysis.

Traditionally, whole program analysis techniques have been looked down upon compared to their incremental cousins. However, as the modern computing environment has evolved, many of the reason for favoring incremental analysis techniques have evaporated. In part due to the open source movement, we live in a world where source code is universally available. Even large commercial libraries ship as source code. The vast majority of applications that benefit from static compilation and type inference have all their source code available for analysis at compile time. For those limited cases where programs must integrate with pre-compiled libraries, the mechanism Starkiller uses for interfacing with foreign code (see Section 2.3) can be readily adapted.

Starkiller's type inference algorithm is flow-insensitive, meaning that it assigns types to variables in the input program source. In contrast, flow-sensitive algorithms assign types to individual variable accesses. In theory, flow-insensitive algorithms are less precise than their flow-sensitive counterparts. However, they are also much simpler to implement. To see how flow-insensitivity can hamper precision, consider

the code in Figure 2-1. The variable `num` is assigned an integer value and then later a floating point value. Flow-insensitive algorithms like the one Starkiller uses combine both of these writes together; consequently both reads of `num` in the calls to `doSomething` see `num` as having a type set of integer, float. A flow-sensitive algorithm would have been smart enough to segregate the two writes so that the first call to `doSomething` would see `num` as having a typeset of integer while the second call would see `num` as having a type set of float. The increased precision has obvious implications for generating faster code.

```
num = 4 # an integer
doSomething(num)
num = 3.14159 # a float
doSomething(num)
```

Figure 2-1: The trouble with flow-insensitive analysis

Starkiller could be augmented to perform flow-sensitive analysis by converting input source programs into Single Static Assignment (SSA) form. In this form, variables are systematically renamed so that each read corresponds to exactly one write. Another approach would involve calculating the set of reachable definitions at every variable read point in the program. Starkiller does not use either of these approaches because they only solve trivial cases like the example shown in Figure 2-1 while being unable to handle more serious sensitivity precision losses. These algorithms are unsuitable for handling anything more complex than multiple writes to the same variable in one scope. Yet the only way that problem can occur is for programmers to reuse the same variable for different purposes. Introducing significant costs simply to provide for better inference of "bad" code is untenable. Moreover, neither SSA nor reachable definitions can handle attribute accesses easily, especially not in the presence of threads, generators, or continuations.

2.1.1 Nodes and Constraints

Starkiller's type inference algorithm works by constructing a dataflow network for types that models the runtime behavior of values in an input program. This dataflow network consists of nodes linked together by constraints. Of course, since Starkiller only has access to information available at compile time, it must make conservative approximations. This means that Starkiller will occasionally infer overly broad type sets for an expression, but it also means that it will never fail to infer the existence of a type that appears at runtime.

Each variable or expression in the input program is modeled by a node that contains a set of possible types that expression can achieve at runtime. Initially, all sets are empty, except for constant expressions, which are initialized to contain the appropriate constant type. Nodes are in turn connected with constraints that model data flow between them. A constraint is a unidirectional link that forces the receiving

node's type set to always contain at least the elements of the sender node's type set. In other words, constraints impose a subset relationship between the pairs of nodes they connect. As a result, types flow along constraints. When a node receives a new type, it adds the type to its type set and promptly dispatches that type to all other nodes connected to it. The algorithm continues until type flow has stabilized.

The preceding discussion raises a question as to how different elements in input program source code generate nodes and constraints. We examine the simplest case, an assignment statement presently and defer more substantiative cases for the following sections. Consider the assignment statement `x = exp` which binds the expression `exp` to the name `x`. Starkiller processes this statement by building a node for `x` if one is not already present and building a constraint from the node corresponding to `exp` to the node corresponding to `x`. That constraint ensures that any types in `exp`'s type set eventually propagate to `x`'s type set.

```
x = 3
y = x
z = y
z = 4.3
```

Figure 2-2: Assignment in action.

As an example, consider the source code in Figure 2-2 and the corresponding constraint network shown in Figure 2-3. Initially, Starkiller creates nodes with empty type sets for the variables `x`, `y`, and `z`. It also creates nodes for the constant expressions 3 and 4.3. However, those two nodes have type set that consist of either the integer type or float type respectively. The assignments indicated dictate that Starkiller place constraints between 3 and `x`, `x` and `y`, `y` and `z`, and 4.3 and `z`. As a result, once processing has completed, the type set for nodes `x` and `y` will contain exactly one element: the integer type. The type set for node `z` will contain two types: the integer type and the float type.

It is important to realize that there are a variety of nodes, each with different behavior. The simplest nodes are variable nodes whose is behavior is as described above. There are more complicated nodes for function calls, function definitions, class definitions and instance definitions. Another important feature of the system is that constraints can be named. This allows a function call node to distinguish between constraints representing different arguments, for example.

2.1.2 Functions

We now examine how Starkiller performs type inference for code that defines or calls Python functions. But first, we review Python semantics regarding function definitions and calls. In Python, functions are first class objects and function definitions are imperative statements that are evaluated at runtime. Consequently, the definition shown in Figure 2-4 serves to create a function object and bind it to the name

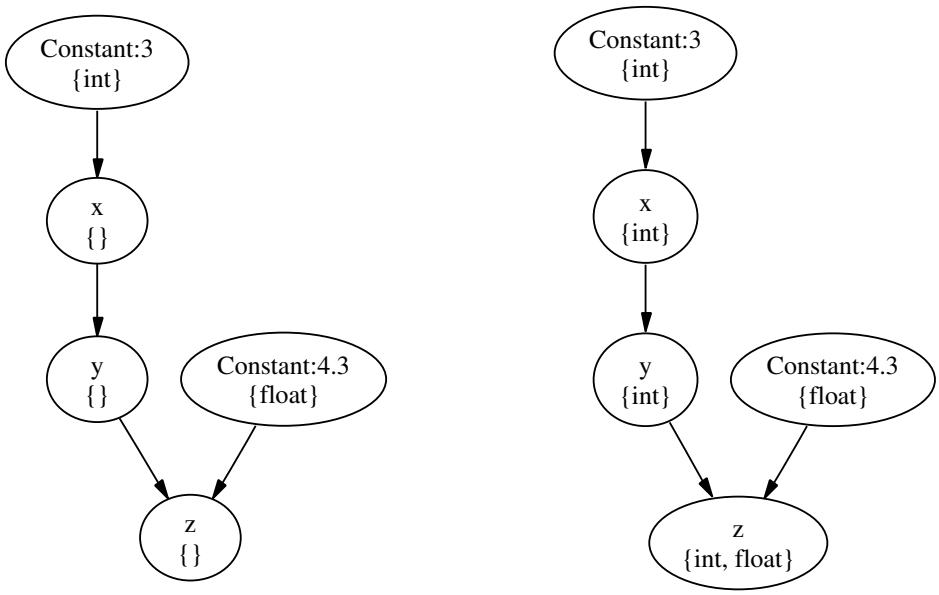

Figure 2-3: Constraint network for the code shown in Figure 2-2 before (left) and after (right) type propagation has converged.

"factorial" when encountered at runtime. Because definitions are evaluated at runtime, the same function definition can easily be instantiated into different function objects. This point is illustrated by the source code in Figure 2-5. Successive calls to `makeAdder` will return distinct function objects since the definition of `add` triggers the creation of a new function object for each invocation of `makeAdder`. Python functions can have default arguments; any such expressions are evaluated once at function definition time. This feature has important implications for type inference.

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n - 1)

x = factorial(5)
y = factorial(3.14)
```

Figure 2-4: The factorial function.

Upon encountering a function definition, Starkiller creates a special function definition node that encapsulates the code associated with the definition. It also creates a node representing the function's name and a constraint from the definition node to the name node. The definition node generates function type objects which then become part of the variable node's type set. In this way, Starkiller models the runtime behavior of function definition as assignment of a newly created function object. Starkiller

```
def makeAdder(a):
    def add(b):
        return a + b
    return add
```

Figure 2-5: A nested function.

also creates constraints from any default argument expressions to the definition node, so that as information about the types of default arguments reaches the definition node, it can produce updated function types that incorporate that information.

When Starkiller finds a function call, it creates a special function call node. It then creates a constraint from the variable node associated with the name of the function being called to the call node. It also creates constraints from each of the nodes associated with the actual arguments to the call node as well. Finally, Starkiller creates a constraint from the call node to wherever the return value of the function call lies. This constraint is used by the call node to transmit the type of the function's return value.

The call node responds to incoming types by taking the Cartesian product over the list of sets of callee types and argument types. The result is a list of a monomorphic types where the first entry is the type of a callee and successive entries represent the argument types. For each monomorphic entry, the function call node attempts to find a matching template. Templates are unique instantiations of the node and constraint graph associated with a single function's source code. Each template associates a monomorphic type for each of the function's arguments. Thus, for any function, there could be many templates that differ only in the argument types. Templates are shared across all call sites of a particular function, so each function in principle only needs to be analyzed once for each monomorphic call signature. If the call node can find a matching template, it adds a constraint from that template's return node to itself so that it can propagate return value types to its caller. If no template exists, the call node creates one, building a set of nodes and constraints.

Starkiller does slightly more than the preceding discussion indicates in order to properly handle lexical scoping. Because nested definitions have access to variables defined in their enclosing scopes, function types must depend on the types of out-of-scope variables referenced by the function. In other words, function types must effectively include the types of all variables they reference that they do not define. For the example code in Figure 2-5, this means that the type of the function `add` incorporates the type of the argument `n`. One benefit of this approach is that function types are completely self contained: they include all information needed to perform inference at a function call site. There are no hidden "lexical pointers" that connect function calls back to the original defining context.

In this way, out-of-scope variables are treated like default arguments. When analyzing functions, Starkiller keeps track of which names are read by each function and which names are written to. This information is used to statically determine a

list of out-of-scope dependencies for each function. Starkiller compiles these nonlocal dependencies recursively, so if a function's lexical parent does not define a nonlocal name that function references, then that name is nonlocal for the parent as well. At function definition time, the types of all nonlocal names are immediately available. Starkiller adds constraints from each nonlocal name to the function definition node, which in turn produces function types that incorporate the types of nonlocal names being referenced. This process is somewhat akin to a common optimization technique for functional languages known as lambda lifting.

The inference algorithm described so far has difficulties analyzing recursive functions. Consider the factorial function shown in Figure 2-4. When Starkiller encounters the definition, it creates a definition node and a name node and links them together. However, because factorial is recursive, it references itself and lists its own name as a nonlocal dependency. Starkiller thus adds a constraint from the name node for factorial to its definition node. Since factorial is a nonlocal name for itself, the function type generated by the definition node must include the type of factorial itself. But now we have generated a cycle. The definition node produces a function type which flows to the name node and then back to the definition node as a nonlocal reference type. As a result, the definition node performs CPA on its list of nonlocal and default argument type sets and produces a new function type that includes that old function type. Rinse, wash, and repeat.

Starkiller solves this problem by detecting the cycle and stopping it. When a definition node sees a new type being sent for a nonlocal name or default argument, it checks to see if that type or any type encapsulated by it is one of the types the definition node itself has generated it. If the incoming type contains a type generated by the definition node, there must be a cycle, and the definition node ignores it. This architecture is designed to alleviate the problems that plagued Agesen's CPA, namely an inability to precisely analyze recursive functions without becoming vulnerable to recursive customization. While not perfect, Starkiller's approach has significant improvements over the heuristics Agesen employed to detect and stop recursive customization.

2.1.3 Classes and Objects

Having explored how Starkiller handles functions, we now turn our attention to how it handles Python's object system. As before, we briefly review Python's semantics before diving into Starkiller's type inference algorithm.

Classes in Python are defined using an imperative statement in much the same way functions are. Within a class body, variables defined are class specific, that is, they are accessible by all instances of the class. Functions defined inside the class are methods. Unlike a function definition, the body of a class definition is executed when the class definition is first encountered. The result of a class definition is the creation of a new class object and the binding of that object to its name. Once created, a class' methods and attributes can be modified or added, even after instances have already been created for it. Class definitions explicitly include an ordered list of base classes. This list can be modified at any time.

Class instances are created by calling the class as a function. This produces an instance as a result and also calls the class' constructor method with the calling arguments as a side effect. As with classes, instances can have attributes added or modified at any time simply by assigning to them. Attribute references using the `self` parameter are first resolved in the instance, then in the class, and then in the class' base classes. Attribute write always occur in the instance; writing to the class or one of its base classes is permitted via explicit naming (i.e., `self.age = 3` for instance attributes versus `self.__class__.age = 3` for class writes). When an attribute lookup fails for the instance and the class, the base classes are perused in the order indicated by a depth first traversal of the inverted base class tree. Attribute accesses in the source code generate GetAttribute and SetAttribute nodes in the constraint network. These nodes have constraints to them indicating what object should be queried, and, for SetAttribute nodes, what the new value should be. The attribute name is a constant string encoded upon construction.

Upon encountering a class definitions Starkiller creates a variable node to hold their name, a new class definition node to generate appropriate class types and a constraint linking the two together. It also creates constraints to the definition node from the nodes corresponding to the listed base classes as well as any nonlocal references. The definition node maintains type sets for each base class and nonlocal reference and takes the Cartesian product of those sets to generate a list of monomorphic types. These monomorphic types packaged as class types and dispatched from the definition node. Reading or writing a class attribute is simple: when a class type reaches a get attribute operation, the attribute type can be read directly from the class type. If it doesn't exist, then no type is returned. When a class type reaches a set attribute call, it does not change its state. Class types are immutable and once created cannot be modified. Instead, they contain a reference back to the definition node that produced them. Rather than change their state, they inform the definition node of the state change, and if necessary, the definition node generates new class types incorporating that change which eventually propagate throughout the system.

When a class type reaches a function call node, it creates an instance definition node at the same place. Much like the class definition node, the instance definition node acts as the repository of the instance's polymorphic type state. The instance definition node generates monomorphic instance types that contain a single type for every attribute defined by applying the cartesian product algorithm over its polymorphic type state. It also creates a new function call node to represent the call to the class constructor, and adds constraints from the original function call to this new one to simulate argument passing. The class constructor is extracted using a GetAttribute node so that if later information adds a new constructor or new base class, all possible constructors will be called.

Because instance attributes shadow class attributes, precision can be lost. For example, a common design scenario is to encode the initial value of an attribute as a class variable, but have instances write newer values as instance attributes directly. This technique can make the initialization logic simpler and cleaner since the instance always has a correct value available.

When an instance type reaches a GetAttribute node, the attribute name is looked

up in the instance type itself, its class type, and its' class type's base classes. All types found as a result of these lookup operations are returned to the GetAttribute node. This parallel attribute lookup suggests a possible loss of precision since, at most, only one lookup result will be passed to the GetAttribute node at runtime. However, this loss in precision is necessary to properly deal with the fact that instance attributes shadow class attributes with the same name. However, because attribute writes must be explicitly qualified, writes do not impose a precision loss. When an instance type reaches a SetAttribute node, it informs its instance definition node of the resulting type change, but it does not change. Like class types, instance types are immutable. Note that Python classes can override the attribute read and write functions with custom methods. This means that, in addition to the process described above, Starkiller must create a call for the `__getattr__` method whenever that method exists and the attribute cannot be found. Calls to the `__setattr__` method must be invoked whenever it exists as well.

Instance method calls require some explanation. Python parses a method call like `inst.method(1,2)` into an AST that looks something like CallFunction(GetAttr(inst, 'method'), (1, 2)). In other words, a bound method is first extracted from the instance using ordinary attribute lookup and that bound method object is then called just like a normal function. Bound methods are first class object, just like functions. The instance attribute machinery in Starkiller packages both the instance type and the method type together into a bound method type that is returned to the GetAttribute node. This behavior only happens when the attribute name is found inside a class or base class and that resulting attribute type is a function.

2.2 Advanced Language Features

2.2.1 Operators

Python allows individual classes to define methods that specify the behavior of overloaded operators. For example, a class that wanted to implement the addition operator would defined a method named `__add__` that takes the instance and the other object being added as parameters. While operator calls internally reduce to method calls, their semantics introduce sufficient complexity so as to preclude treating them as mere syntactic sugar for method calls. This complexity stems from Python's coercion rules, and their long storied history of informal specification, special casing, and evolutionary change.

These coercion rules specify which method will actually be executed for a particular operator call given a pair of operands. For an operator `op` the coercion rules specify selection of a method named `__op__` defined in the left operand. If such a method is not defined, the rules mandate that a method named `__rop__` defined for the right operand be used, where the r prefix indicates right associativity. If that method is undefined, the implementation throws a NotImplemented exception. There are further complications, such as special case rules for operator precedence of new style classes where the right operand is an instance of a proper subclass of

the left operand. In-place operators (e.g., +=) introduce further complexity since a method `__iop__` is searched first and used without coercion if found, but, if that method is not defined, execution falls back to a combination of the standard operator (`__op__`) and assignment. A complete copy of the coercion rules in all their hideous soul devouring glory can be found in Section 3.3.8 of [48].

Starkiller handles operator calls by creating an operator node that has constraints from the operand types and that generates the result type for that operation. Internally, the operator node performs CPA on the operand types just like a function call node, but unlike a function call node, it enumerates the resulting monomorphic operand type lists and uses the coercion rules to determine which method from which operand should be called for each monomorphic pair of operands. Having found the method, it builds a get attribute node to extract the correct method from the correct operand and a function call node to call that method so as to determine the result type of the operation. Since the result types of all of these function call nodes point back to the operator node using a named constraint, the operator node can easily generate a return type for the value of the operation expression.

2.2.2 Exceptions

Python provides structured exceptions not unlike those found in Java and C++. One notable difference is that functions and methods do not (and can not) declare a list of exceptions they can potentially throw. Exceptions can be any Python object, although exception comparisons in the catch clause of try/except statements are based on class and subclass comparisons. In other words, an exception will be caught by an exception handler only if it is a an instance of a class or subclass of the handler's exception target.

Starkiller's type inference algorithm refuses to deal with exceptions in any way. This is because exception handling is complex and delicate and in many ways, represents a corner case in language implementation. In addition, with one small exception, exception handling provides no useful type information to a flow insensitive algorithm. Traditionally, Python exceptions have been understood as a slow mechanism for transferring control. This experience is also true in C++, Starkiller's target language. As a result, exception handling invariably occurs outside the critical path, and thus sees little if any benefit to aggressive optimization.

The one instance where exception handling provides necessary type information to a type inferencer is for programs that make use of named exceptions. Such programs use a variant of the except clause in a try/catch statement to bind the captured exception to a local variable and make it available to exception handler. Starkiller cannot perform complete type inference on programs that use this feature since it does not process type information for exceptions. One simple alternative would be to use the type of the exception handler's target as a surrogate for the type of the exception actually caught, but this approach would violently break language compatibility. The crux of the problem is that the exception object that must be bound locally is generated in a later call frame, and we cannot determine that without exerting significant effort to track the movement of exception objects across call frames.

While the current implementation of Starkiller does not process exceptions, later versions easily could using the strategy outlined below. Doing so would involve using the static call graph, of which Starkiller has ready access to a conservative approximation. Each function call node already maintains constraints to itself from the return nodes of each template it makes use of. Exception handling could work in much the same manner, where function and method bodies maintained a hidden variable to record the types of all uncaught exceptions and passed them back to their callers using a named constraint in the same way that they pass return values to their callers.

In order to deal with the fact that a given function can contain multiple possibly nested try/except clauses, we introduce an exception node for each of them and build constraints between them over which exceptions flow based on their scoping relationships. Function call nodes have named constraints emanating from themselves to the most closely scoped exception node. When they encounter an exception type from one of their callee templates, they pass it along to the nearest exception node, which may make it available in a local binding if the exception matches the target or may propagate it onward to the next outer exception scope. This design supports all Python exception semantics while ensuring that inference remains complete, even in the face of named exceptions. Its major drawback is that precision for exception expressions may be reduced in some cases, but it should be more than adequate to improve upon the current Python implementation's performance.

2.2.3 Iterators and Generators

Recent Python releases have introduced iterators [50] and generators [32] into the language. Their design was heavily influenced by the Sather [29] and Icon [18] programming languages, where such objects play a vital role. Iterators are objects that follow a standard protocol whereby they provide clients with a stream of intermediate values using a `next` method and signal the end of iteration by raising a `StopIteration` exception. For–loops act naturally over iterators. Generators are a particular type of iterator object that make use of the yield keyword. They allow functions to naturally suspend their entire execution state while providing intermediate values to their callers without terminating.

As with exceptions, Starkiller provides no support currently for generators and iterators. Due to the tight relationship between iterators and exceptions, support for the latter necessitates support for the former. Nevertheless, once Starkiller's type inference algorithm properly supports exceptions, it can be easily extended to support iterators and generators. Iterator support can be added by modifying how Starkiller analyzes for–loops to check for and make use of `__iter__` and `next` methods when present. Generators can be supported by recognizing functions that contain the yield keyword and ensuring that any calls to them result in a generator object that implements the iterator protocol. Each invocation of the next method should be equivalent to executing the body of the generator, with the yield statement being used to return values to the caller instead of the return statement as used in traditional functions.

2.2.4 Modules

The Python module system allows statements of the form `from email.encoders import encode_base64` and `import socket`. When using the former, objects named on the import statement are made immediately available to the importing module after the import statement has executed. When using the later form, module objects are bound to a variable with their name. A module's contents can be accessed using named attributes, such as `socket.gethostbyname`. Starkiller reflects these semantics by representing each module with a type. Note that the term module can refer to a Python source file or an extension module with a corresponding Starkiller type description.

Like function types, module types have templates associated with them, but unlike function types, there is only one template associated with each module type. Upon encountering an import statement, Starkiller locates the needed module and checks whether it has been analyzed before. Since Python caches module imports in order to ensure that each module is executed only once, Starkiller replicates this behavior by only analyzing a module once and thereafter referencing its template. For named imports, where some or all of the module's contents are imported into another module, Starkiller creates a variable node in the importing module for each name being imported. It then builds constraints from the original variable node in the imported module template to the newly created variable node of the same name in the importing module.

2.3 Foreign Code Interactions

Because it must perform type inference on programs that use both native Python source code as well as Python extensions written in other languages (typically C, C++, or Fortran), Starkiller is faced with a serious problem. Moreover, these external languages and the APIs they use to interact with Python are sufficiently complex so as to make type inference infeasible. In order to resolve this impasse, extension module authors are expected to write simple descriptions of the run time type behavior of their modules. These External Type Descriptions are written in a language that Starkiller specifically provides for just this purpose. In this section, we begin by exploring the External Type Description language. We conclude by examining the external type description for one particular module, the `__builtins__` module, which defines all of Python's core types and functions.

2.3.1 External Type Descriptions

External Type Descriptions are Python classes that subclass an ExternalType class provided by Starkiller. That base class provides its subclasses with a stylized interface into Starkiller's internals. There are also ExternalFunction, ExternalMethod and ExternalModule base classes provided to complement ExtensionType. The power of the extension type description system is difficult to understate; it allows extension authors to "plug into" Starkiller's type inference machinery and make their own

code part of that machinery. The relationship between Lisp and its macro system is analogous to the relationship between Starkiller and its external type description system.

Extension type description authors write description classes that describe the run time type behavior of their extension types. These classes are subclassed from ExtensionType and can use its getState and addState methods to manipulate the polymorphic state of that particular instance of the extension type. These description classes include methods that describe the behavior of the actual methods implemented by the extension type. For example, when an extension type instance's `append` method is called, Starkiller will invoke the corresponding method in the extension type to determine the method call return type. The append method is presented not just with the monomorphic argument types of the method call, but also the result node that it will be directed to and the monomorphic state of extension type instance. The method can retrieve or add new type state to the extension type or any other type visible to it and it can also use the callFunction method to simulate arbitrary function calls. This feature can be used to model extension functions that return result types that depend on calling one of their arguments, such as `map`. In fact, the method can apply arbitrary transformations to the constraint network.

External types are treated in much the same way as class instances. In particular, their internal state is described completely by a mapping from state names to polymorphic variables. For example, if an external list type has method names `append` and `insert`, it will have a state element for each with their corresponding name and a type set consisting of a single external method type. As with class instances, when external type instances reach a get attribute node and are asked for a method, they generate a bound method type that encapsulates both the external type instance as well as the method name. When this bound external method type reaches a call function node in the network (i.e., when someone attempts to execute an external type's method), the corresponding extension type method is called and asked to supply a return type for the operation. In lieu of supplying a return type, the method has the option of creating a link to the destination node for the return type so that it can be filled persistently. Polymorphic external type instances are associated with instance definition nodes that generate appropriate monomorphic external type instance state types that propagate through the network. Because their architecture mirrors that of native Python classes, external types see the same level of precision that native Python classes do.

The External Type Description Language is as follows. Type descriptions are short files written in the Python language. A type description describes the behavior of an extension module. Within each type description file are individual descriptions for the external functions and types provided by the module. External functions are represented by a class that must inherit from `ExternalFunctionInstance`, a base class that Starkiller provides. This class should define a method named `call` that takes the following arguments:

1. `self`, the instance of that external function being called

2. `resultNode`, the node in the constraint network to which the result of this

function call is being directed

3. `monoState`, the monomorphic state of the function object

4. the actual arguments of the function

Note that the actual arguments of the function can include default and keyword arguments just like a normal Python function, which is why it is described using Python functions. During type inference, when native Python code calls an external function object, the types of the arguments associated with that call are passed after the first three arguments described above. As usual, Starkiller performs CPA on the argument types of external function calls, so even if the argument types are polymorphic, the external function instance will only have to deal with monomorphic argument types at any one time. The return value of the `call` method is a set of types corresponding to the type of the value returned by the actual external function. Note that Python's builtin datatypes can be referenced from the `basicTypes` module, which is automatically imported. In addition to simply calling an external function, Python can also read and write the function's attributes. Consequently, if extension authors define methods with the same arguments as `call` but named `attr_read_NAME` and `attr_write_NAME`, Starkiller will treat attribute reads and writes of the function object as if they were properties and invoke the appropriate methods of the instance type.

External types are similar to external functions. Each external type is represented by a single class that must subclass `ExternalTypeInstance`, a base class that Starkiller provides. They handle attribute accesses in the same way that external function instances do. But instead of a single `call` method, they define one method for each method implemented by the external type. These methods have the same calling signature as `call`, but are named `method_NAME` instead. A list of the methods defined should be provided in the class attribute `definedMethods`.

2.3.2 The builtins Module Type Description

Python includes a special module called `__builtins__` that encompasses all the basic types and functions accessible by default. It includes named constructors for the basic types of machine integers, double precision floating point numbers, long integers, lists, tuples, and dictionaries. These basic data types are completely pervasive in all real Python programs; in fact, certain syntactic constructs use them implicitly. For example, tuple unpacking happens automatically in assignment statements like `a, b = foo()`. Moreover, a class' base class list is represented as a tuple of classes stored in an attribute named `__bases__`. In order to properly analyze programs using these basic types, Starkiller includes an external type description for the `__builtins__` module. Those type descriptions will be described briefly, both to illustrate the issues involved in writing a proper external type description and to showcase the limits of inferencer precision when using basic datatypes. It is important to note that for a given external type, there can be many different external type descriptions that differ significantly in precision and run time inferencer performance.

Contrary to their name, Python lists are vector–like data structures capable of holding any type of element while supporting fast appending at the end and constant time element writing and retrieval. Starkiller provides an external type description for lists that maximizes precision in the common case while degrading gracefully in the face of operations that necessitate imprecision. The crux of the problem is that while we want to be able to associate precise types with individual slots of a list, there are some list operations that modify the lists' type state in a way that is impossible for a flow insensitive inferencer to trace. For example, consider the case of a list initialized with the statement `L = [1, 2.2, z]`. Starkiller has enough information on the list to determine that the type of the first element of the list will be an integer. However, if at some later point in the program, `L.sort()` is called, we can no longer make that guarantee since sort swaps the list's elements in place.

Consequently, the external type description for lists maintains a state variable indicating whether that particular list has been "tainted" by such operations. The list keeps a state variable which describes the type set of each slot and an extra state variable describing the type set of all slots combined. Reading from a particular slot of an untainted list yields the type set of that slot, while a slot read from a tainted list yields the combined type of all slots. Lists initially are untainted, but can become tainted when they become subject to a tainting operation, like the sort method described above. When that happens, additional types may be propagated through the constraint network to reflect the fact that previously single slot reads may actually refer to the contents of any slot. This protocol works well with CPA because it never violates the monotonicity principal: the set of types produced by reading a list item is always (partially) correct, but may not always be complete. Adding types is safe, but removing them is not.

Tuples use a very similar mechanism, namely maintaining state for each element. Of course, since tuples are immutable, they need not deal with tainting operations. Dictionaries use a similar method to associate key and value types together. Astute readers will notice one serious problem with the scheme described above: Starkiller deals in types, not values, but in order properly associate individual list indices with unique types, it must necessarily keep track of values. In general, Starkiller tracks types, however, in a limited number of special cases, it will actually process constant values. The two special cases are attribute names and indexing operations. Failing to provide special case value handling of constant attribute names would make Starkiller so imprecise as to be useless: the type of any attribute referenced would have to be equivalent to the combined type set off all attributes. Indexing using the `__getitem___` and `__setitem__` methods plays just as serious role. Without special case support for constant index expressions, operations involving implicit tuple packing and unpacking would lose a great deal of precision. Unfortunately, such operations are utterly pervasive: they are simply too convenient not to use. Python code generally includes both constant and variable forms of both attribute access and indexing. Thus we see expressions like `x.attr` and `y[2]` in addition to expressions like `getattr(x, userInput())` and `y[z*2 - 1]`.

A simplified form of the external type description that Starkiller supplies for lists is shown in Figure 2-6. This version does not implement the "tainted" behavior

```
class homogenousList(ExternalTypeInstance):
    definedMethods = ('__getitem__', '__setitem__', '__len__',
                      'append', 'count', 'sort', 'reverse')

    def method___getitem__(self, resultNode, monoState, itemIndex):
        return self.getState(monoState, 'elementTypes')

    def method___setitem__(self, resultNode, monoState,
                           itemIndex, itemValue):
        self.addState('elementTypes', itemValue)
        return Set((basicTypes.NoneType,))

    def method___len__(self, resultNode, monoState):
        return Set((basicTypes.IntType,))

    def method_append(self, resultNode, monoState, itemToAdd):
        # arguments after self are each a single monomorphic type
        self.addState('elementTypes', itemToAdd)
        # return value is a tset
        return Set((basicTypes.NoneType,))

    def method_count(self, resultNode, monoState, item):
        return Set((basicTypes.IntType,))

    def method_sort(self, resultNode, monoState):
        return Set((basicTypes.NoneType,))

    def method_reverse(self, resultNode, monoState):
        return Set((basicTypes.NoneType,))
```

Figure 2-6: An external type description for the list type from the builtins module

needed to provide isolated per-slot behavior; instead, the types of all slots in the list are commingled together into a single state variable called `elementTypes`. The type description is simply a subclass of `ExternalTypeInstance` with a set of named methods that correspond to the methods of the external type. When Starkiller encounters a method call for an external type, it calls the corresponding method in the external type description, passing the method call's monomorphic argument types as parameters. In addition, Starkiller also passes the monomorphic state of that instance as well as the result node where the method call's return value is being sent. The former allows the type description code to interrogate the state variables associated with the instance being called while the latter allow the type description code to build constraints directly to the result node. Such constraint building is required in cases where the result type of a method call comes directly from a normal Python function

call or global state since the result could change as constraint propagation evolves. Note that many of the methods defined are rather simple: they return None or an integer and nothing else. This example demonstrates how simple most extensions are from a type inference perspective.

The challenge for extension type description authors lies in designing extension types descriptions that can make good use of constant information when available while degrading gracefully in the face of variable attribute and index accesses, all the while maintaining the monotonicity invariant that Starkiller relies upon.

2.4 Known Problems and Limits

Starkiller's type inferencer has a number of problems that need to be resolved before it be used in production environments.

2.4.1 Poor Handling of Megamorphic Variables

The type inferencer currently assumes that it will never encounter any megamorphic variables. These are variables for which the number of possible concrete types is very large. Such variables are problematic because the cartesian product of a list of megamorphic variables can easily grow far beyond the memory capacity of a single machine. For an example of where this problem can occur even absent truly pathological cases, consider HTMLgen, a component library that simplifies the creation of dynamic HTML pages. Each tag in HTML has a corresponding class in HTMLgen; HTML documents are constructed by creating instances of the different tags as needed and stringing them together in trees. HTMLgen has about 75 different classes, all of which are essentially interchangeable as potential document tags. Thus a function that took three tags as arguments would have 421,875 unique templates associated with it, far more than could be constructed or analyzed in reasonable amount of time.

Starkiller currently does nothing to recognize and deal specifically with megamorphic variables; in the hypothetical example above, it will happily begin analyzing 421,875 different templates although the user is unlikely to ever allow it to finish. In order to reliably analyze programs that use large component libraries like HTMLgen, Starkiller's type inferencer must be modified to recognize when the cardinality of a function call's argument's type set exceeds some threshold. Similar modifications must be made when dealing with data polymorphism in class, instance, and external type descriptors. The response to such megamorphism must be the abandonment of precision; in other words, we trade execution time and memory space for precision. As Agesen points out in [1], the precision lost when dealing specially with megamorphic variables is often a useless precision since such variables see no benefit from the cartesian product because they represent a collection of interchangeable types that must be dynamically dispatched in any event.

Recognizing megamorphism is insufficient by itself. Starkiller's type inferencer must also adapt by not including megamorphic variables when taking the cartesian product. Moreover, the inferencer must be modified to accept megamorphic sets of

types in many places where it expects monomorphic types. The transition will be a difficult one. In the same vein, the compiler must be modified to deal appropriately with megamorphism so as not to generate an enormous number of templates.

2.4.2 Template Instance Shadowing

Template instance shadowing is a complex problem that may result in the loss of precision in some cases. The crux of the matter is that the stateful nature of instance state types does not interact well with Agesen's stateless monotonic model of CPA. The problem is as follows. Ordinarily, each template should create a new instance type so that the different templates do not contaminate a single instance type. This occurs normally when the instance is created in the polymorphic function. But if the instance is created in another function that returns it to the polymorphic function, the same instance type will be shared by all templates of the polymorphic function.

```
class a:
    pass

def f():
    return a()

def g(arg):
    w = f()
    w.attr = arg

x = g(1)
y = g(3.14159)
```

Figure 2-7: Template instance shadowing in action.

This problem will be made clearer with an example, shown in Figure 2-7. Assume a function `f` returns an instance of class `a`. Another function `g` acquires an instance of `a` by calling `f`; `g` takes a single argument and assigns that argument as an attribute of the newly return instance. If `g` is then used to generate two different instances of `a` named `x` and `y`, the result will be two objects that have different types associated with the attribute `attr`. However, Starkiller will conclude that in both `x` and `y`, the attribute `attr` has a polymorphic type that contains both integer and float. This is because exactly one instance type is returned by the cached template for the function `f`, so the different templates of `g` must share it. Consequently, the types associated with different templates of `g` commingle.

It remains unclear to what extent template instance shadowing will prove detrimental to precision in practice. It is possible that constructs associated with template instance shadowing are sufficiently infrequent so as to make its impact negligible. In the event that template instance shadowing is responsible for a significant loss of precision, there are several possible techniques that can be employed to eliminate

it. One such technique involves intercepting instance state types that traverse constraints which cross template boundaries. We only do this for types that originate in definition nodes associated with a sibling or child scope of the destination template's scope. During interception, we replace the outgoing type with a copy of itself by creating a new definition node whose state is seeded by the state of the outgoing type.

2.4.3 Partial Evaluation

Starkiller's implementation for dealing with attributes is rather inelegant. Essentially, support for explicit constant attribute names has been hard coded into the type inferencer as a special case, since the program source contains them directly. A more elegant approach would have been to treat all attribute accesses like invocations of the `getattr` function. Since this function cannot know statically what attribute name it will look up, it returns the types of all attributes together. Unfortunately, named attribute references are far too common for this approach to have been practical; its application would have rendered all non-trivial programs a morass of imprecision. However, it would be desirable to replace Starkiller's special case logic for dealing with attribute access with a more generalized partial evaluation framework, since constant explicit attribute access is simply a special case of partial evaluation. This approach would provide other benefits as well, such as improving precision when analyzing some common programming idioms.

One such idiom describes how to build a class that deals with many other client classes without expanding those client classes. It works by implementing one method for each client class to be examined and uses Python's introspection features to select the correct method to call based on one of the client class' data attributes. For example, consider Figure 2-8 which portrays a stereotypical implementation of the Visitor design pattern [14] used to implement part of a compiler. The compiler defines classes for all possible nodes of an abstract syntax tree, including nodes to represent constants, if statements, and for–loops. If we wanted to write code that would operate on instances of all those nodes without touching the original classes, we might build a visitor class as shown in Figure 2-8. This class uses introspection to determine the class name of any instance presented to it and then uses that name to construct an appropriate method name to process that instance. This is a well known idiom in the Python community.

Starkiller handles code like this, but not well. Because it performs no partial evaluation, it assumes that the `getattr` call can return any value that is an attribute of instances of the visitor class, when in fact, it can only return attributes whose names begin with the string "visit". Starkiller's conservative assumption leads to imprecision which in turn hinders performance and increases generated code size since extra type checks and error handling code must be synthesized to deal with the invocation of other attributes besides the visit methods. A more general solution would obviate the need to check for impossible combinations that cannot arise in practice.

```
class Constant(Node):
    pass

class If(Node):
    pass

class For(Node):
    pass


class visit:
    def processNode(self, node):
        return getattr(self, 'visit' + node.__class__.__name__)(node)

    def visitConstant(self, node):
        pass

    def visitIf(self, node):
        pass

    def visitFor(self, node):
        pass
```

Figure 2-8: The visitor design pattern in Python.

Chapter 3

Compiler Design

While Starkiller's type inferencer is absolutely vital and in many ways its most novel component, the compiler is its heart. Starkiller's compiler processes as input Abstract Syntax Trees (ASTs) representing Python source code annotated with type inference information provided by the type inferencer. These annotated ASTs are used to generate C++ source code for an equivalent program. This equivalent program can then be compiled to machine code using a common, off the shelf C++ compiler such as g++. Between processing input ASTs and generating output C++ code, Starkiller's compiler has ample opportunity to perform optimization passes ensuring that the resulting code runs as fast as possible. Starkiller's compiler encompasses not only the code generation component, but also the runtime support library that compiled output programs rely upon.

The design of a source to source translator like Starkiller's compiler presents the designer with a vast multitude of design decisions to solve. In contrast to the type inferencer, the compiler affords far more leeway in design and implementation. These design decisions include basic questions such as what target language the compiler should generate code for and how data structures should be translated. Other significant questions include how individual language features should be translated as well as what kinds of optimizations should be performed. This chapter examines all these questions and the answers that Starkiller's compiler represents. Note that while a very primitive version of this compiler has been built, much of what follows is a design for a system that has not yet been implemented.

3.1 Core Design Decisions

3.1.1 Target Language Selection Criteria

In order to realize the full performance benefits implied by Starkiller's type inferencer, the compiler must generate code in a suitable language. In particular, that language must enable the creation of high performance code. In other words, appropriate target languages will have powerful optimizers that benefit from the last few decades of research in compiler optimization. This criteria essentially requires the use of a statically typed target language, a requirement bolstered by the fact that Starkiller is

able to determine static types for most expressions in input programs. Fine grained control over memory layout and allocation are also required in order to make the best use of the static type and size information Starkiller provides. Given Python's historical predilections as a glue language, interoperability with existing C, C++ and Fortran code bases is an absolute must. While not strictly necessary, platform portability is a significant benefit, as is the availability of multiple implementations of the target compiler.

The above criterion severely constrain the choice of target languages, but other features in Starkiller's design reduce those constraints. For example, since the output program is generated by machine, the target language need not be concise nor need it have a pleasant syntax. In fact, the target language can be brutally painful for a human being to write code in without hampering Starkiller's objectives in the slightest.

3.1.2 Target Language Selection Rational

In selecting a target language, several possibilities were considered, specifically C, C++, Java, Common Lisp, Scheme, Haskell, and variants of ML. Common Lisp and Scheme were eliminated early on because they are dynamically typed and lack high performance, low cost implementations that run on a variety of platforms. Haskell and the ML languages were removed from consideration because they do not give target programs fine grained control over memory management and structure layout in addition to failing to interoperate well with existing foreign code bases. Java was eliminated on the basis of its poor performance. Only C and C++ survived the first round of elimination.

Python's original VM implementation was written in ANSI C with particular emphasis on portability, although other implementations have been written in Java [21] and Microsoft's common language runtime [22]. Like Python, many dynamic languages have virtual machines or compiler output written in C. C offers the benefit of simplicity and universality: C code runs as close to hardware as anyone concerned with portability dares, thus ensuring that C compilers are available for every platform known to man. Moreover, since C++ compilers also compile C programs, integrating with existing C and C++ code bases is a simple matter.

In contrast to C's spartan simplicity, C++ is large and complex. It is a testament to how unwieldy C++ has become that nearly 20 years elapsed before C++ compilers that complied with the ISO C++ standard were actually produced. Nevertheless, C++ offers a number of benefits over C that are difficult to ignore. The availability of classes makes implementation more convenient and the presence of references improves safety while offering the compiler greater opportunities for optimization. C++'s native exception handling system provides an important substrate needed in all Python programs that would be difficult and time consuming to replicate in C. Moreover, the generic programming facilities in general and the Standard Template Library in particular greatly simplify the implementation of Starkiller's polymorphic templates. The STL offers other benefits in providing a library of fast, safe, and debugged containers and algorithms that would have to be built by hand in a C im-

plementation. One shudders at the comparison between fundamentally unsafe string handling in the C standard library versus its counterpart in the STL string class.

While C++ does not enjoy the incredible platform support of C, support for the language has matured greatly in the last few years; the advent of g++ as a serious compiler that actually implements (almost) the entire standard means that there is at least a decent C++ compiler available for (almost) every 32-bit platform one can find. Much has been made about the performance differences between C and C++. Empirical results suggest that C++ compilers generate code that is just as fast as C compilers when given similar code [11]. Significant performance disparities arise when programmers use C++ constructs that are impossible to optimize well, such as virtual methods [11]. However, Starkiller simply does not use such features. Other C++ language features such as exceptions and templates may increase the size of executable, but have no direct impact on run time performance [11].

Another benefit that C++ provides is that its generic programming facilities make it substantially easier to drop in libraries in a standardized manner. For an example of where this might be useful, consider CPython's specialized memory allocator for integers. This allocator has been shown to substantially improve performance in comparison to the generic object allocator [22]. By using C++ as the target language, Starkiller gains the ability to drop in the Boost memory pool allocators [9] with only a few lines of code. Of course, since Starkiller uses unboxed arithmetic whenever possible, optimized integer allocation is not nearly as important, but there are substantial benefits to be had for other object types.

3.1.3 Intermediate Language

Starkiller uses the results generated by the type inferencer to annotate an AST representation of the input program in order to produce code in an intermediate language. The compiler only processes intermediate language ASTs. The intermediate language effectively comprises a strongly typed version of Python: variables are annotated with concrete types. A variable's type is either a monomorphic type or a polymorphic type that comprises a known sequence of monomorphic types. Monomorphic types include the basic Python types like integers, floats, long integers, lists, tuples, and dictionaries as well as classes and class instances. A monomorphic type can have polymorphic attributes, as in the case of an instance that has a polymorphic attribute. It is important to note that the types used in the intermediate language do not necessarily correspond to those used by the type inferencer; the process by which the type inferencer's results are used to generate code in the intermediate language is described in more detail below. In particular, there are some things that the type inferencer does to improve analysis precision that are of little or no use to compiled code. For example, the inferencer generates monomorphic instance state types to represent instances with polymorphic state attributes. The compiler, however, simply deals with instance types that have polymorphic attributes directly since one cannot change an instance's type at runtime to accommodate attribute values with new types.

Types that are executable, such as functions, modules, and class definitions, contain a type map. This mapping describes the types of the variables defined in the

executable's body. Function types have an executable section and type map for every realized template while class and module types have exactly one template. Classes and modules can thus have polymorphic attributes whereas functions can only have monomorphic arguments. Functions that have polymorphic default arguments are represented as several different function types, one for each of the cartesian product of the polymorphic default argument types.

Types in the intermediate language can be tagged with attributes that describe particular features of their behavior. Two such attributes are VarGetAttr and VarSetAttr which indicate whether a type may be exposed to the `getattr` or `setattr` functions respectively. This information can be easily determined once type inference is complete and is vitally important for generation of fast code.

Note that before the type inferencer even sees the input program, it is first transformed into a flattened form in which expressions cannot be nested. Compound expressions are broken down into primitive expressions that are assigned unique names. The compound expressions are then built referencing the newly introduced variable names. For example, the Python statement `x = sqrt((a + b) * c)` will be transformed into the following sequence of statements, where `u0` and `u1` are new unique variables introduced into the program shown in Figure 3-1. This flattening differs from single static assignment in that it ignores control flow and thus introduces no choice point operators. In addition to simplifying implementation of the type inferencer, flat input programs are required for some of the fast polymorphic dispatch mechanism discussed in Section 3.2.2.

```
u0 = a + b
u1 = u0 * c
x = sqrt(u1)
```

Figure 3-1: Transformed segment of Python source code.

3.1.4 Data Model and Memory Management

A compiler's business consists of two principal activities: control flow and data flow. While the compiler's handling of control flow is discussed at great length in Section 3.2, we describe its handling of data presently. Basic Python types, such as integers, floating point numbers, and strings are mapped to equivalent C++ classes defined in the Starkiller runtime library. Thus, for example, Starkiller's run time support library includes a C++ class named `skInteger` that contains a C++ machine integer and implements all the basic operations described by Python's standard numeric interface [48, Section 3.3.7]. These wrapper classes bridge the chasm between native C++ types like and `int`, `double`, and `std::string` and their Pythonic siblings.

Numbers are stack allocated and passed by value; they are thus always unboxed. All other values are heap allocated and passed by reference using pointers. Heap

allocating most objects raises the issue of how and when those objects are ever deallocated. The current implementation of Python uses a reference counting scheme whereby each object maintains a count of how many other objects point to it. Since reference counting cannot handle cycles of object references, CPython augments basic reference counting with a periodic garbage collection phase designed to explicitly break reference cycles. Starkiller abandons reference counting entirely in favor of conservative garbage collection. It uses the Boehm garbage collection library [5]. In contrast to implementing a reference counting scheme, this approach dramatically reduces both object size and development time while increasing performance. The problem with using garbage collection is that it reduces portability since the Boehm implementation requires explicit support from the operating system. In practice, this is not a problem as the Boehm collector has been ported to most popular operating systems and platforms.

Opting for conservative garbage collection, at least initially, helps to mitigate other problems, specifically, a design flaw in C++ involving exception handling and memory allocation. The problem is that when an exception is thrown from within an object's constructor, the object's destructor is never called. The object's memory is safely deallocated, but any cleanup actions that need to be done and are normally the province of the destructor never occur. A critical example is the case of an object that allocates a sub object off the heap in its constructor and then deletes that memory in its destructor: if an exception occurs in the constructor after the call to new, the sub object's will never be deleted, resulting in a memory leak. One solution to this problem is to use smart pointers to describe heap allocated objects. When the object is deallocated after an exception is thrown during its constructor, the smart pointer will be deleted which will trigger its destructor which will, in turn, deallocate the heap allocated sub object. The use of smart pointers such as `std::auto_ptr` is a messy "solution" that brings with it additional complexities and inefficiencies. In contrast, garbage collection solves this problem cleanly, without placing undue burdens on programmers.

It has long been an article of faith among C++ programmers that in order to maximize performance, objects should be stack allocated whenever possible. More specifically, they believe that objects should *only* be heap allocated if life cycle rules mandate their continued existence outside the scope of their creation. Their rational is that in most implementations, stack allocation requires a constant increment to the stack pointer on function entry while free store allocation requires searching through a free list to locate a block of memory that is sufficiently large and then marking the free list as needed. Stack space is more likely to be in the cache and can be accessed using constant offsets from the stack pointer that can be determined at compile time in contrast to free store references which typically require pointer dereferences from a distant and likely uncached region of memory. Deallocation is greatly simplified since it consists of nothing more than decrementing the stack pointer once in contrast to far more expensive strategies such as the continuous arithmetic implied by reference counting or the cache poisoning pointer walking required by even the best garbage collectors. Finally, stack allocation may allow the optimizer to generate better code because it is easier to avoid aliasing in some cases when using stack allocation in

contrast to free store references. Yet another benefit of stack allocation is that it embodies the Resource Acquisition is Initialization paradigm, which greatly simplifies development when using C++'s baroque exception handling scheme. While often cited as a significant benefit, this feature does not motivate

Starkiller will have an extra optimization pass that performs escape analysis to determine which objects have the possibility of escaping the scope in which they are created [38]. Objects which can be shown not to escape their creation scope can be allocated on the stack. Doing so not only speeds up allocation, but also deallocation since there is less free store memory used that the garbage collector must track and reclaim. Starkiller's intermediate language is well suited to traditional escape analysis algorithms, such as those described in [15, 8].

3.2 Basic Language Features

3.2.1 Arithmetic Operations

Arithmetic operations present a critical problem for Starkiller because Python operators must handle overflow. This behavior is not described in the language specification, but it has been enforced in all versions of Python. Note that while the most recent releases of Python introduce new behavior that automatically coerces integer objects to long integer objects in response to overflow, the underlying implementation still needs to determine when overflow occurs. Previous versions of Python responded to overflow by throwing an exception. Traditional Python implementations have checked for overflow by performing explicit checks after each arithmetic operation. For example, when CPython calculates the product `z` of integers `x` and `y`, it checks for overflow by verifying that `(z ^ x) < 0 && (z ^ y) < 0`. Since Starkiller must also detect overflow, it could conceivably use the exact same technique. However, following each arithmetic operation with an explicit branch is incredibly slow. Beyond the cost of extra instructions, this method severely limits performance by inducing extra pipeline stalls and using up valuable branch prediction slots.

Alternative techniques for detecting overflow include using special instructions such as `JO` (jump if overflow bit is set) and using processor exceptions to detect overflow and inform the application automatically. CPython tolerates the performance penalty incurred by explicit checks rather than use either of these techniques for three reasons:

Portability and simplicity Explicit checks written in C provide identical semantics on all architectures in contrast to fragile tests written in assembler that are unique to each architecture and possibly operating system.

Foreign code Both techniques described above require that the processor state be kept consistent, but this is very difficult in cases where Python is either embedded in a larger C program or extended with foreign code. In either case, Python cannot reasonably guarantee that code outside the Python core will keep the processor state consistent during calls back into the Python core. For

example, if an extension author is careless about resetting the carry flag, when Python relies on that flag to detect overflow, havoc will result. Preemptive multi threading makes the problem even worse since control could transfer between application threads between individual instructions.

Other performance losses dominate The performance penalty is small compared with many other overheads associated with the virtual machine. However, since Starkiller eliminates many of those conventional sources of overhead, the performance loss of checking for overflow after each arithmetic operation becomes more substantial.

Instead of accepting such a severe performance penalty, Starkiller opts for an alternative mechanism to detect overflow. By setting the appropriate CPU and FPU control words, Starkiller generated code can instruct the processor to generate an exception in response to overflow conditions. This exception forces the processor to transfer control to the operating system. The OS in turn informs the application using Unix signals (specifically, `SIGFPE`). Starkiller generated code includes a signal handler that responds appropriately to `SIGFPE` signals by throwing a C++ exception. That C++ exception could encapsulate a Python `OverflowError` exception. Alternatively, silent coercion to long integers could occur if blocks of arithmetic operations were wrapped in C++ try–catch statements that re-executed the entire block using long integer types in response to an overflow exception. This mechanism allows Starkiller to generate code that responds appropriately to integer overflow without resorting to explicit checks on each and every arithmetic operation.

Note that the mechanism just described is not completely portable; at the very least, it requires an IEEE 754 compliant floating point unit and may impose additional requirements on the choice of processor architecture, operating system, standard C library, and C++ compiler. This is because while many CPUs and Unix-like operating systems support overflow detection through hardware exceptions and Unix signals as described above, it is not required by any standard. In fact, according to the C and C++ standards, program behavior after signed integer overflow occurs is undefined and implementation dependent. Moreover, the ability to successfully throw C++ exceptions from within a signal handler is a rather delicate process; it is partially supported in at least some C++ compilers. Empirical investigations suggest that this mechanism will work correctly on Intel IA-32 and IA-64 architectures running GNU/Linux in combination with recent versions of g++.

The relative complexity of the process described above may require additional work or introduce additional points of failure. For example, under some conditions, Intel microprocessors do report an overflow condition during the instruction that triggers it, but during the following instruction. In addition, signal delivery is not guaranteed in all cases and can fail under some conditions in some operating systems. Finally, usage of machine exceptions in this manner complicates the life of extension authors while arguably improving overall application safety. This problem is not as significant for Starkiller as it is for CPython because there are far fewer reasons to include numerically intensive foreign code extensions in Starkiller.

3.2.2 Functions

Despite its apparent simplicity, a strategy for efficiently implementing Python functions in C++ while maintaining the original Python semantics has proven difficult to develop. Nevertheless, such a strategy has been developed and we now describe it in some detail. In part, this extra detail stems from the fact that functions are emblematic of a host of implementation issues seen in other areas of the compiler. Another reason why functions merit detailed description is that they are crucially used in building higher level constructs. For example, class methods are really just functions bound to a class object. When selected from a class object, they behave precisely as ordinary functions would, because they are ordinary functions. As another example, consider that the case of dynamic dispatch in Python programs is really a special case of polymorphic function dispatch that is described below.

It is tempting to believe that Starkiller could compile Python functions into equivalent C++ function definitions. Unfortunately, for a variety of reasons, such simple translation is impossible in the general case. One reason is that Python definitions are imperative statements while C++ definitions have declarative semantics. Another reason is that Python functions are first class objects in their own right which can have attributes associated with them. Yet another reason is that Python functions can be nested and can close over variables defined in the scope of their lexical parent. In theory, some of these problems can be worked around. For example, gcc offers an language extension that supports nested scopes. However, on closer examination, one can see that that extension is unsuitable for Starkiller's purposes because it provides no support for inner functions escaping their parent. Consequently, Starkiller implements Python functions as C++ objects. In other words, each function definition is associated with a class; thus, defining a function in Python triggers the construction of an instance of that definition's C++ class and binds the newly created function object to a variable with function's name.

This approach allows Starkiller to model all the semantics of Python functions. Nevertheless, there are additional subtleties to Starkiller's translation of function definitions. One of those subtleties is the handling of default arguments. In Python, a function definition can specify default values for some of the function's arguments. Such values are evaluated exactly once when the function definition is encountered. Starkiller stores these default values as member variables of the function's implementation class. Their precise values are assigned when the function object is created by the class constructor. Since the "same" function can be defined many times over if it is defined in the body of another function, each definition creates a new instance of that function with new values for the default arguments.

Closures

Another vital detail relating to Starkiller's handling of functions is the compilation mechanism used for closures and lexical scoping. Starkiller can detect which variables that a function defines are referenced by other functions defined within that scope. Whereas variables are ordinarily allocated on the stack, Starkiller allocates such inner

scope variables from a Mini Stack Frame (MST). Note that variables consist of either primitive objects such as integers or floats that are stored by value or a pointer to a heap allocated object. An MST is a custom block of memory which contains these variable references or primitive values. Functions that define variables referenced in inner scopes create an MST object on entry and refer to all such variables using the MST. Functions that reference variables defined in their lexical parents take in the address of the MST which defines those variables as a parameter of their construction. These pointers to MSTs are stored as member variables in each function object and enable the function instance to transparently reference variables defined in their parents' scope. This implementation strategy relies heavily on one crucial feature of the Python language, namely the fact that variables cannot be written to from inner scopes. In other words, variables defined in an outer scope can be read from, but not assigned to. The one exception to this rule is global variables referenced using the `global` keyword. However, since they are allocated statically in the module, such variables can be easily dealt with as a special case.

The rational behind the MST is that in order for both the defining and inner functions to see the same set of variables, those variables must be allocated in space that is available to both of them. However, because an inner function can (and often will) persist even after its lexical parent has returned, Starkiller cannot use space on the stack to store these shared references since that space will be reclaimed when the defining function returns. In the most general case, the best that Starkiller can do is to allocate the space for such variables from the free store. Garbage collection will ensure that the MST will not be reclaimed as long as either the defining function or some inner function persists and refers to it. Given the constraints described above, MSTs are a relative efficient implementation technique for dealing with closures. If a function is defined N layers deep, then it will have at most $N + 1$ MST's. Because function definition nesting is in general infrequent and almost always very shallow when it does occur, most inner functions will have only one or two MSTs. Moreover, all functions defined in or below a given scope will reference the same MST, minimizing storage overhead. Another performance benefit is that MSTs are only needed for variables referenced from an inner scope; variables that are not referenced from an inner scope do not pay the performance penalty imposed by the MST. The primary performance costs associated with MSTs are the need to allocate from the free store on function entry and the extra cost of indirect references for MST variables.

Starkiller attempts to minimize these costs in two way. The first method exploits the fact that variables defined in the top level of a module are created exactly once. Python's module system has the property that module objects are created exactly once, and that subsequent imports of the same module simply return references to the existing module object. Because of this semantic guarantee, module variables can be placed directly within the module object itself, and module objects can be allocated statically. Since the majority of out-of-scope accesses will likely be reads of objects defined in the top level of modules, this optimization should significantly reduce the need for MSTs. Module objects will not be accessible through a constant offset of the stack pointer, but they will be the next best thing since their address will be a known constant. The second technique that Starkiller uses to minimize

the detrimental performance impact imposed by MSTs is a limited form of escape analysis. Although not as sophisticated as the analysis described in Section 3.1.4, escape analysis can prove very beneficial. If an inner function can be shown to never escape from its lexical parent, then the variables referenced by that inner function can be allocated normally on the stack, bypassing the need for an MST altogether. Instead of an MST, inner functions must take a reference or pointer to the stack allocated variables as a parameter of their constructor.

Fast Polymorphic Dispatch

When performing type inference, Starkiller analyzes functions independently for every possible set of monomorphic arguments. Such individual instances of functions are called templates. Starkiller's compiler mirrors this approach by generating different copies of function objects' call method. These copies are distinguished by their argument types. This duplication of code provides the compiler with similar benefits as the corresponding duplication of analysis provides the type inferencer. In cases where a function call's argument types are known statically, the compiler can inline the appropriate call method. Beyond simple inlining lie a whole host of compiler optimizations that only become possible when the code to be executed is known at compile time.

Creating new templates for each monomorphic argument type list that a function may encounter can lead to significantly increased code size. An alternative approach would create only one version of the function body which accepted polymorphic arguments. Unfortunately, the loss of precision incurred by completely polymorphic function bodies would spread to any functions called by the function in question. Starkiller's approach gives the compiler as much knowledge as possible to support optimization while maintaining maximal precision since the function's body has been specialized for the particular monomorphic types. Consequently, functions called by the function in question can be inlined and optimized as well. Even if a particular function benefits little from having monomorphic versions of itself available, functions called by that function can be better optimized since their argument types are more likely to be monomorphic. As with type inference, the process of maximizing precision in compilation leads to a virtuous cycle in which monomorphic functions call other functions that can in turn benefit from monomorphic arguments. Nevertheless, in may be advantagous to bound template creation by the compiler even though the type inferencer operates under no such restriction. The benefit here is that the type inferencer can still work without losing any precision, but the compiler need not generate an enormous amount of code that presents little if any performance benefit.

Of course, this virtuous cycle can only occur when all of a function's argument types are known statically. While that is often the case, there are times when Starkiller's type inferencer cannot statically reduce an argument's type set to exactly one entry. Such cases are either an artifact of the conservative approximations the type inferencer makes, or represent the intent of the programmer. In any event, Starkiller must insert code at the site of function calls involving polymorphic arguments to determine the argument types at runtime and call the appropriate version

of the function. This problem is similar to the one faced by object oriented languages that perform dynamic dispatch. In fact, because more than one argument type may be polymorphic, this problem is equivalent to that faced by languages that support multimethods, such as Dylan [35] or CLOS [12]. This is a simple problem to solve unless one cares about performance. Typical implementations use the argument types as indices into a table of function pointers. Because of the performance properties of modern CPU architectures, compilers that use this approach generate code that performs poorly.

This performance loss is caused by the need to perform an indirect branch through the lookup table as well as the optimization opportunities forsaken by remaining ignorant about precisely which piece of code will be executed at the call site. The trend in architecture evolution has been toward longer and longer instruction pipelines which impose correspondingly stronger performance penalties on instruction sequences that trigger pipeline stalls, such as indirect branches. While modern architectures rely on a combination of branch prediction and speculative execution to achieve high performance in the face of code laden with run time choice points, neither of these strategies provides significant benefit to code that uses indirect branches extensively. This problem has proven to be such a serious impediment to high performance in common object oriented languages like C++ and Java, that microprocessor vendors have begun adding indirect branch prediction units to their most recent chips [13]. The success of the new indirect branch prediction units is too recent a development to evaluate, but the difficulty of the problem bodes poorly. Beyond the problems with pipeline stalls and optimization opportunities foiled by dynamic code selection, the size of the dispatch tables also plays a role in reducing performance. This occurs because the tables tend to be large and thus place additional stress on the memory cache. Fortunately, this particular problem can be ameliorated to a large extent using simple compression techniques as described in [25].

In some ways, Starkiller has a slightly easier problem to solve than other statically typed object oriented languages because it does not have to worry about new classes and types being introduced at runtime. Nevertheless, the problem remains daunting. Starkiller thus offers several different approaches for dealing with polymorphic argument dispatch in the hopes that the relative performance merits of each can be characterized and compared on a variety of different architectures. The three options offered are traditional dispatch tables, branch ladders, and signature hash dispatch. Traditional dispatch mechanisms suffer from the problems described above, but are likely to be the optimal choice in cases where there are many function templates that may be called from a given call site or if the templates are very large or if the function code takes a significant amount of time to execute (as in the case of a short function that makes system calls). In these cases, the overhead of the indirect dispatch and function call will be negligible compared to the execution time of the function itself and the costs of implementing the other two polymorphic dispatch mechanisms will likely be prohibitive. Once the performance of the different algorithms has been characterized on different architectures, Starkiller will include a heuristic for selecting the correct dispatch mechanism at each polymorphic call site. One possible heuristic is to use traditional vtable dispatch for megamorphic calls in which there are more than

eight possible templates while using signature hashing for cases where there are between three and eight possible templates and using branch ladders for cases in which one is trying to select between only two possible templates. This flexible multimodal dispatch agent is a simplified version of what is described in [3].

Branch ladders are akin to a technique described in [52] where dynamic dispatch is replaced with a binary search through the receiver types followed by a direct jump to the appropriate method. For cases in which there are only a few applicable methods, this technique has been shown to provide a moderate performance improvement over traditional dispatch tables. Instead of binary search, branch ladders perform dispatch by testing each polymorphic argument's type code against every possible type code that argument can achieve at runtime. Once an argument's type code has been found, dispatch on the next polymorphic argument type begins. After all polymorphic arguments have been discovered, the proper method is called. This dispatch technique thus generates a set of nested if tests where the depth of nesting is equal to the number of arguments with polymorphic types and the number of branches at any particular depth is equal to the number of possible types that the corresponding argument may achieve. This approach has the benefit of working with the processor's branch prediction unit rather than against it. However, since branch prediction units have relatively small tables (often less than 2000 entries) and relatively poor resolution (often only a few bits), this technique will perform poorly in cases where there is either a large number of polymorphic arguments or where the degree of polymorphism is high. Even if it is only used with function calls that have a small number of minimally polymorphic arguments, it may still perform poorly if used so pervasively that the branch prediction unit's storage tables overflow. An alternative implementation uses switch statements instead of if statements. For some combinations of compiler and processor architecture, switch statements may prove more efficient.

The last technique that Starkiller can use to perform polymorphic function dispatch is signature hashing. This actually refers to a family of techniques that all rely upon quickly combining the type codes of the polymorphic arguments of a function call into a single number, the signature, and using the that signature to perform dispatch. In effect, this process amounts to hashing the type codes of the polymorphic arguments. Once the signature has been calculated, there are several different mechanisms that can employ it to perform dispatch. The first method is a variant of the branch ladder technique described above. The primary difference though is that there is no nesting of branches: there is only one layer of tests, rather than one layer per polymorphic argument. Each test compares the signature against a constant value and if it succeeds, calls a particular specialized version of the function call method. Alternatively, the signature hash can be used as input to a perfect hash that takes generates indices into a computed goto array for dispatch. In this scheme, each function call generates a series of blocks, one for each possible monomorphic argument combination. Each block is comprised of a label, a specialized function call, and a goto statement that transfers control past the end of the last block. The addresses of the individual block labels are stored in a static array using g++'s "labels as lvalues" extension. Upon reaching a function call with polymorphic arguments, Starkiller inserts code to calculate the signature hash and then use that number to calculate

an index into the block label table. Once an index is known, jumping to the appropriate block using g++'s computed goto extension is simple. Starkiller converts the signature hash to the appropriate block label table index using a specially designed, minimal perfect hash function.

Perfect hash functions are hash functions that guarantee no collisions assuming the set of all possible keys is known statically. In this case, the perfect hash maps a set of integers (the signature hashes) to another set of integers (the indices of the block label table). Minimal refers to the fact that the hash function generated maps a set of N keys onto the integer range $[0, n-1]$ with no more than N elements. In general, a different perfect hash function must be generated for each function call site since most function call sites will have different polymorphic argument types. This duplication is no cause for concern because creating the perfect hash functions is a fast process and because the generated hash functions are so small they will almost certainly be inlined in any event, thus negating any savings from calling a single perfect hash from multiple call sites. The algorithm Starkiller uses to generate perfect hashes is the same one found in [23]. For small input sets (those with 8 or fewer keys), it generates hash functions that are only a few instructions long. These functions are simple combinations of logical shifts, negations, and addition operations. They use no other data except an occasional constant integer and the input signature hash. These hash functions run very quickly since they are composed entirely of ALU operations that only reference a single register and a few immediate values.

As an example, consider the perfect hash function generated for signature hashes drawn from the set $(432, 32, 0, 49, 97, 34)$. If the signature hash is an integer variable named *sig*, the resulting hash function is $(((sig >> 1) + (sig >> 8))\&7)$. In summary, the signature hash maps the types of the polymorphic arguments to a single number. A perfect hash function is then used to map that single number to an index into the block label table. The runtime jumps to the address at that index of the table. This architecture may seem convoluted, but with the exception of the computed goto, all the operations that comprise it are very fast on modern architectures. The computed goto normally compiles down to a single jump to register instruction, which is not fast, but is faster than many of the alternatives. However, the most significant benefit is the ability to inline small functions and perform many of the other basic optimizations that are easily foiled by dynamic dispatch.

In theory, one can avoid the overhead associated with the block label table by using the perfect hash function to generate block offsets directly from signature hashes. In practice, this process is rather difficult since the precise offsets needed cannot be known until the platform assembler is run on the compiler's output. The core of the problem is that g++ does not calculate such offsets directly: they remain in the assembler input as constants. The problem can be remedied either by directly generating machine code or by patching the assembler output of the compiler before it is assembled to machine code, but the difficulty of either of these two options compels us to leave such modifications for another day. In any event, the benefit appears minimal.

3.2.3 Classes and Class Instances

Like function definitions and unlike classes in C++ and many other languages, class definitions in Python are imperative rather than declarative statements. Consequently, they can appear anywhere in a program, including the body of a function. Starkiller treats class definitions and function definitions in a similar manner. Class definitions have a C++ class that describes the state of the class and stores its attributes. Upon encountering class definition, Starkiller generates code to construct and initialize an instance of that class's corresponding C++ class. The terminology is somewhat confusing; suffice it to say that every Python class object is an instance of a C++ class. The result is assigned to a variable with the same name as that specified in the class definition. This process will be made clearer with an example. Consider the Python source code in Figure 3-2 that depicts a simple class definition. Starkiller compiles that code into something like the C++ source code shown in Figure 3-3. The definition for class `a` occurs inside function `f`, so Starkiller create a new instance of the class object in the body of `f` on each invocation of `f`. Once the class object is constructed and bound to the class' name, `a`, an instance is constructed by calling the class object. This instance is then returned from `f`.

```
def f():
    class a:
        pass
    inst = a()
    return inst
```

Figure 3-2: A simple Python class definition.

Like Starkiller and the rest of this thesis, this section applies to classic classes only. New style classes [47] must be handled differently, but in many ways pose a simpler problem than classic classes. Besides using a different mechanism to search the class inheritance tree, new style classes differ principally in their ability to explicitly and statically describe their attribute set and limit some forms of dynamism. Since classic classes will continue to be used for a long time to come (especially if Starkiller proves successful in optimizing their performance), Starkiller focuses on implementing them first.

Class objects and instances at the most basic level can be thought of as collections of named attributes; these collections are not unlike dictionaries but with a more convenient syntax. Consequently, any compilation process that supports classes must expressly support attribute access. However, attribute access is also needed for function objects as well as module objects. Starkiller internally uses a protocol for reading and writing statically visible attributes. If a given object has an attribute named X, Starkiller will add methods to the corresponding C++ class named set_X and get_X. In the typical case, these methods will be inlined. However, in more complex cases, these methods can be used to implement specialized behavior such as

```
class f_class;
static f_class* f;

class a_class;
class a_inst_class;

class f_class
{
public:
  a_inst_class* operator()()
  {
    a_class* a;
    a = new a_class();

    a_inst_class* inst;
    inst = a();
    return inst;
  }
};

...

f = new f_class();
```

Figure 3-3: The equivalent code generated by Starkiller for the definition shown in Figure 3-2.

reading from either the instance attribute directly or the instance's class' attribute depending on whether the instance's attribute has been defined yet. Of course, this protocol is only useful for attributes whose names are statically known. Attributes whose names are only apparent at runtime are called dynamic attributes. Starkiller uses special techniques for dealing with them, as described in Section 3.2.3.

Beyond being a simple bag of attributes, One of the most notable features of Python class objects is that they support multiple inheritance. More specifically, every class definition specifies a list of base classes from which the class being defined inherits. The ordered set of base classes associated with a class object can be inspected and even replaced by referencing the attribute `__bases__`. In Python, inheritance means nothing more than different rules for attribute lookup. Specifically, searching for a particular attribute of a class object will cause the class object to ask its base classes for that attribute if it is not defined. Base classes are searched in depth first order, from left to right. The first match found is used; multiple matchings are ignored. Assignment must be explicitly qualified: the only way to assign an attribute to a class (outside of its definition) is to explicitly state the class name as in `c.attr = 4` for a class named `c`. Class attribute writes are thus always unambiguous,

which is not the case for class attribute reads in the presence of inheritance.

Starkiller implements static class attribute semantics by providing special code for the get_ methods of the C++ classes used to implement class objects. In cases where a class inherits from other classes that either define the same name or have been tagged with the VarSetAttr attribute by the type inferencer, this special lookup code walks up the inheritance tree in depth first, left to right order and returns the first matching attribute value it finds. However, since inheritance is far less frequently used in Python than in other popular object oriented languages, and since name shadowing is infrequently used even when inheritance is used, this case represents a rare occurrence. In most cases, the attribute lookup method will simply return the value bound to the class object directly. Since attribute writes are always explicitly qualified, the set_ methods can simply write into the class object directly without having to deal with inheritance at all.

Class instances are created by calling a class object as if it were a function. Like class objects, instances are bags of attributes, but their behavior differs significantly. For example, while attribute writes modify the instance state directly, attribute reads may not necessarily refer to instance state. If the requested attribute is not defined by an instance, the instance will search for it in its class object, which may in turn search for it in its base classes. Starkiller implements different get_ methods for statically known instance attributes depending on which types define them. If an attribute is known only to the instance, the get_ method simply return the instance state for that attribute. If it is defined only in the class object or one of its base classes, the get_ method simply returns the result of calling the same get_ method on the class object. In the unfortunate case where both the instance and the class or a base class define the same attribute, the get_ method returns the instance value if it has been defined; otherwise, it delegates the request to the class object.

Instances distinguish themselves in other ways. The most significant is their treatment of methods. When an instance retrieves an attribute from its class object and that attribute happens to be a function, a special transformation is applied. Instead of returning the function object directly, the instance returns a bound method object that packages the instance itself and the function object together. Note that this process does not happen when one directly references the class object: attribute lookup works as usual there. The bound method object is a callable object that calls the function object using the list of arguments supplied after prepending it with the instance object. This is the primary manner in which Python implements traditional object oriented programming. Starkiller generates appropriate bound method classes for each function object that can become a class attribute. Instance get_ methods that refer to attribute names that may return a bound method object create them as needed. However, in the common case where a method name is referenced and then immediately called, Starkiller optimizes away the temporary bound method object and directly calls the function object that would be returned. Of course, this optimization only occurs when the method attribute type is monomorphic.

As in other languages, Python's object model includes explicit support for constructors and destructors. These are overloaded methods defined by a class that bear the names `__init__` and `__del__`. Despite the name constructor, `__init__`

methods really only perform object initialization: they are presented with an already constructed, but empty, object instance just like any other method would be. In a similar manner, the `__del__` method actually implements object finalization, not destruction. Instance construction is implemented with an internal construct method defined in class objects. This method creates an instance of the C++ class implementing the type of instance needed and returns it after calling the associated `__init__` method if it is available. Destructors are somewhat more difficult to implement. Since intrinsic types like integers and floats are passed by value, their destruction is managed automatically when their scope of use exits, as long as their C++ destructor calls the equivalent Python `__del__` method. All other objects however, are deleted by garbage collector. The Boehm collector will also call the appropriate C++ destructor for classes that inherit from the base class gc_cleanup, as all Starkiller classes that may implement finalizers do. The Boehm collector provides guarantees equivalent to those provided by the Python language specification in pathological cases, such as when given a cycle of objects that include finalizers.

In addition the behavior described above, instances differ in one other key way. They always have a special attribute named `__class__` that refers to the instance's class object. This attribute can be modified or inspected at run time, allowing arbitrary client code to change the class associated with a particular instance at any time. Consequently, Starkiller provides the necessary get_ and set_ methods when the type inferencer indicates they may be used. The vast majority of objects, however, will never require explicit inspection or assignment of their `__class__` variable.

There are several cases where get_ methods rely on knowing whether a particular object has actually defined an attribute. For example, when the same attribute is defined by an instance and its class object, the value associated with the instance object should not be returned unless it has been assigned. As another example, consider the case where a base class and its derived class define an attribute with the same name. In all such cases, Starkiller adds boolean member variables to the object's C++ class definition that indicate whether a particular attribute has been defined for the object. These member variables default to FALSE, but are assigned a value of true on each call to the corresponding set_ method.

Dynamic Attributes

Dynamic attributes are attributes whose names cannot be determined statically. Class objects and instances that have a VarGetAttr attribute require special code for dynamic attribute lookup and retrieval. Such lookup operations return values of a polymorphic type that encompasses the types of all the object's attributes, since a dynamic attribute lookup could, in principle, retrieve any attribute the object has. Since the set of all possible valid keys is known, the lookup operation is implemented using a near minimal perfect hash [23]. This enables very fast retrieval. More specifically, a perfect hash is used that maps each of the attribute name strings to an integer index describing which of the object's attributes have that name. The lookup code thus computes the hash and returns the correct attribute, using either a switch statement or a set of computed goto labels. Unfortunately, when presented with unseen

keys, a perfect hash algorithm will return an incorrect mapping, so the lookup operation must check the incoming key name for equality with the key name generated by the hash. If the two do not match, a runtime `AttributeError` exception is raised, as would be the case in CPython. This extra string check imposes a small additional runtime cost, but is needed to ensure correct operation.

Class objects and instances that have a VarSetAttr attribute cannot implement that functionality with extra code alone as is the case for objects tagged with VarGetAttr. In addition to extra code, objects tagged with VarSetAttr require an internal dictionary that is used to store the values of all the attributes that are not statically known to Starkiller. Statically visible attributes are stored separately in the object as normal. The variable attribute dictionary's keys are strings corresponding to the name of the attribute. Note that for a class `c`, this dictionary is not the object that would be returned by `c.__dict__` since it contains only the dynamic attributes but not statically visible attributes. Starkiller generates a special dynamic setattr method for these objects that assigns new attribute values to either the statically known attributes or the dynamic attribute dictionary. It does so using a very similar algorithm as described above for looking up attributes of objects tagged as VarGetAttr. More specifically, the setattr method first uses a perfect hash to map the incoming attribute name to an attribute index. It then compares the incoming attribute name to the known attribute name associated with that index; if the two names match, that particular attribute is modified. If they do not match, the incoming attribute is added to the dynamic attribute dictionary.

Objects that are tagged with both the VarGetAttr and VarSetAttr attributes inherit the union of behaviors associated with either attribute. They maintain a dynamic attribute dictionary and have a special dynamic setattr method that dispatches assignments to either statically known attributes or the dynamic dictionary as needed. They also have a special dynamic getattr dictionary that searches for attributes amongst the statically known set as well as in the dynamic attribute dictionary.

Starkiller's approach for dealing with VarSetAttr tagged objects makes the best of a bad situation. Since a call to `setattr` could conceivably mutate any attribute, the types of all attributes must be polymorphic and must include at least the types assigned by the `setattr` calls. An alternative design would have made all objects tagged with the VarSetAttr attribute implemented using only a dictionary with no support for separate static attributes. On the surface, this approach appears to provide similar tradeoffs, however, this alternative approach provides slightly fewer avenues for optimization. Consider the case of a class object tagged with VarSetAttr. Any references to the class objects method attributes will necessarily be polluted by the types deriving from the dynamic `setattr`. Because such types will necessarily be polymorphic, method calls for that class cannot be inlined without including expensive dynamic dispatch code.

Starkiller can optimize this problem away with only a slight loss of compliance to the Python language specification by assuming non-callable objects cannot be assigned to method attributes for this class. Consequently, the method attributes can have monomorphic types that do not include the types deriving from the dynamic

`setattr` call. Starkiller will generate a dynamic setattr method for this class that verifies that the key being assigned to is not a method name, and throws an exception if it is. In most cases, the user of a class takes special steps to ensure that their use of the `setattr` function will not modify the value of method attributes. Assuming that is true, this optimization will not change program semantics in user-visible manner. If this assumption is violated and user code really does replace class methods will non-callable objects, then an exception will be raised in the likely event of those methods being called. With this optimization, an exception gets raised, but in a different place and time than would be dictated by the language specification. In the exceedingly unlikely event that the class user has replaced a method attribute but then taken steps to ensure that method will never be called again, this optimization would trigger an exception whereas the language specification indicates that no exception would occur. Regardless of the merits of this particular optimization, the point remains that it is much easier to implement when statically known attributes are separated from the dynamic attribute dictionary.

Since class objects and instances are not implemented as dictionaries in Starkiller, all objects trap requests for the attribute named `__dict__`. The object returned by `c.__dict__` is a proxy object that includes a reference to the original object and implements the standard dictionary interface. It simulates a dictionary that includes all of the elements of the object's dynamic attribute dictionary in addition to the object's statically known attributes.

There are two implementation methods that Starkiller offers for providing this proxy dictionary. The approaches tradeoff object size and speed for strict compatibility with the language specification. In the optimized approach, no space is reserved in the object itself for this `__dict__` proxy; it is created dynamically in response to attribute lookups. Different requests for the dictionary attribute will return different proxy objects, but because their underlying storage is the same, the different proxies will behave identically in all aspects except for one. Because they are different objects, they will test false for identity (even though they test true for equality). In other words, the expression `c.__dict__ is c.__dict__` will return false, whereas the language reference indicates that it should return true. It is unlikely any code exists which relies on such subtle semantics.

Nevertheless, for those individuals who do have code that relies on dictionary identity semantics, or even those unsure as to what assumptions their code relies on, Starkiller provides a compilation option that provides strict compliance with the language reference. In this mode, space is allocated for a dictionary proxy in the object and one proxy object is created during object construction. That proxy is referenced throughout the lifetime of the object. In practice, not all objects need to be bloated in space and construction time with the dictionary proxy. Only those object types that are tagged as VarGetAttr and those types that have constant references to the `__dict__` attribute need an dictionary proxy.

It is important to realize that as a consequence of how Starkiller implements support for `__dict__`, it cannot use simple reference counting alone. This is because when using the conservative `__dict__` implementation described above, many objects will be created with reference cycles automatically. These cycles occur because class

objects and instances maintain a pointer to their dictionary proxy object which in turn keeps a pointer back to them. Both pointers are needed since we must ensure that, even if the object is no longer needed, it will not be deallocated as long as a reference is still being held to its dictionary proxy.

3.3 Advanced Language Features

3.3.1 Exceptions

Observant readers will note that despite the fact that Starkiller compiles Python code to C++, it does not port Python language constructs directly onto their C++ equivalents. This is especially true in the case of functions and objects. In large part, that discrepancy is due to the fact that the C++ function and object models are simply too restrictive to support the full semantics Python demands. In contrast, Starkiller uses C++'s exception system rather directly.

Starkiller defines one exception class that it uses pervasively called `InternalRuntimeError` (IRE). This exception class acts as a wrapper for native Python exceptions as well as internal exceptions in the Starkiller runtime that have no Python equivalent. Python exceptions cannot be handled directly by the C++ exception system because C++ determines whether exception objects can be captured by a catch clause using class identity and inheritance. Python uses the same mechanism. Unfortunately, because Starkiller's implementation of Python objects does not use C++ inheritance to model Python class inheritance relationships, Python exception catching semantics cannot be modeled using C++'s exception handling system without wrapper classes.

All functions and methods that Starkiller generates as well as all of its runtime library routines are declared as capable of throwing instances of IRE and no other C++ exception. Any native C++ functions or methods that can throw exceptions are wrapped in exception handlers when used by Starkiller library components. These handlers trap native C++ exceptions and re raise them as IRE instances. When possible, C++ exceptions are translated into Python equivalent exceptions. For example, if object allocation fails during a call to operator `new` and a `std::bad_alloc` exception is raised, Starkiller's runtime system will catch that exception and re raise it as a Python `MemoryError` exception object. In fact, operator `new` is shadowed with a Starkiller provided version that performs such transparent exception translation automatically.

Within the infrastructure described thus far, there are only two major problems left to resolve: the precise manner in which Starkiller translates try-except-finally statements and raise statements. Try statements in Python are translated into Try-except statements in C++ but with the caveat that the only exception listed in the C++ catch clause is IRE. Within the catch clause, Starkiller generates code to inspect the native Python exception object associated with the instance of IRE just caught. If the Python exception matches the exception classes listed in the corresponding except clause of the Python program, the appropriate exception handling code is executed. Alternatively, if the captured exception does not match, it is reissued for

the next handler to catch and test. Raise statements can be easily handled in this framework. The Python exception being raised is constructed as usual and then packaged in an IRE object which is then thrown using the standard C++ facilities for raising exceptions.

Python try-except statements that include finally clauses are problematic since C++ does not have an equivalent mechanism in its exception handling system. Nevertheless, Starkiller can effectively simulate such a mechanism by judicious code generation. Specifically, when presented with a finally clause, Starkiller inserts the finalization code in two different places. The code is inserted immediately after the try-except statement which ensures that it will be executed in all cases where no exception was thrown or all exceptions thrown were safely caught by this try-except statement. Starkiller also inserts the finalization code inside the exception handler, immediately before the statement needed to re raise the exception. Doing so ensures that even if the exception propagates beyond the current scope, the finalization code will be processed.

One important detail to note is that constructors for Starkiller objects, including functions and classes, are designed to never throw exceptions. Since arbitrary Python code can throw exceptions at any time, Starkiller takes pains to ensure that arbitrary Python code is never executed from within a constructor. For example, the code in the body of a class definition can potentially raise exceptions, but Starkiller does not run such code in the class object's constructor. Instead, it relegates such code to an initialize method that is called after the constructor has finished its work from the same scope in which the class object was created. Essentially, constructors are for only the most basic allocation and setup. The reason Starkiller strives to avoid throwing from constructors is that when exceptions are thrown from a constructor, the object's destructor is never called. This complicates memory management and several other resource management problems sufficiently to make no-throw constructors worth the small extra effort they require.

3.3.2 Modules

Like class objects, module objects are implemented with C++ classes. Like classical function objects, the implementation class for a module objects includes an overloaded function call method which contains the code associated with the module body. Because modules are objects just like classes and instances, they can be passed around and inspected just like other objects.

3.3.3 Foreign Code

Since Starkiller's type inferencer has extensive support for dealing with foreign code, the compiler must provide corresponding facilities. However, the compiler's task is far simpler than the type inferencer's in this regard. CPython uses a standard C API to describe interactions between the virtual machine and foreign extensions. Starkiller's compiler simply has to include an implementation for the interface defined in the

header file `Python.h`. This interface consists of functions, macros, and constants for creating, destroying, and manipulating Python objects.

3.3.4 Iterators and Generators

Iterators require very little support from Starkiller in terms of compilation. The primary issue is recognizing types that support the iterator protocol when compiling for–loops and if available, using that protocol to generate successive values of the loop variable. Generators demand substantially more effort on the compiler's part. A generator must store sufficient information to restore both its data state (i.e., the state of variables) and its control flow state (i.e., where it is in the code). Each generator has a C++ class whose instances act as a repository for that generator's internal state while it is running. the traditional function object associated with a generator definition does nothing but construct a generator state object and return it.

Starkiller ensures that the generator's data state is stored by converting all variables in the generator body into member variables of the generator state object. This transformation guarantees that those variables persist and keep their values even as control flow leaves the generator's code during a yield. Control flow state is stored in an extra member variable that denotes the next location in the generator's body that is to be executed. Ordinarily, this might require an arbitrary instruction address, but since generators only transfer control synchronously using the yield statement, we can do better. If control only leaves the generator at yield statements, then control can only return to the generator at the very beginning of the body (when it is first entered) or immediately after a yield statement. Starkiller adds distinct labels immediately after yield statements. These labels are then stored in the generator state object's next code member variable using g++'s labels as lvalues extension to C++.

The body of the generator is thus placed inside a method of the generator state object called `__next__` that implements the generator protocol [50]. This method is called on each iteration of the generator but the generator's state is stored in member variables between successive calls. Yield statements in the generator body are translated into control state saving operations followed by the equivalent return statements and a distinct label. Saving the control flow state is as simple as assigning the the value of the label immediately following the yield to the next code member variable. That is the last statement executed before the method returns. The first line of the method body is a computed goto statement that jumps to the point in the method body indicated by the next code member variable.

In order for the generator to work on the very first call to `__next__()`, before any yield statements have been called, there must be a label immediately following the computed goto statement demarcating the beginning of the official generator body code. In addition, the next code member variable must be initialized in the generator state object constructor to the value of that start of code label.

Chapter 4

Results

Having completed our tour of Starkiller's design, we now turn to more practical matters, namely the progress that is been made in implementing Starkiller and preliminary benchmarks of its performance.

4.1 Current status

Starkiller consists of two main components, a type inferencer and a compiler. The type inferencer is largely complete; it can interpret and correctly analyze a large variety of Python code. Many of the language features that it cannot currently understand are relatively simple but technically uninteresting constructs that were passed over in the rush to develop a usable inferencer that performed well on difficult inputs. As a result, the type inferencer correctly analysis programs that change an instance's class at runtime as well as programs that change inheritance relationships at runtime but cannot understand programs that use for loops or tuple unpacking.

Starkiller's compiler is at a very early stage of development; it currently lags far behind the type inferencer in terms of both completeness and extensibility. The compiler currently supports a very small subset of the language, consisting of little more than basic arithmetic, simple functions, while loops, conditionals, and only the most primitive forms of IO. Closures and nested functions, classes, modules, exceptions, and generators are completely unimplemented. Despite its limitations, development of the compiler has been enormously useful in illustrating the unique design challenges presented by Starkiller. The wisdom gained thereby will prove invaluable when constructing the next compiler.

4.2 Benchmarks

Starkiller's overriding goal is to make Python programs run faster. A simple benchmark was devised in order to determine whether or not it has met that goal. This effort was hampered by the fact that Starkiller currently only compiles a small subset of Python, even though it can analyze a much larger subset. It would have been preferable to use a standardized and well known benchmark, such as PyStone, but

```
def fact(n):
    if n <= 1:
        return 1
    else:
        return n*fact(n - 1)

def fib(n):
    if n < 3:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)

i = 0
while i < 100000:
    q = fact(12)
    q2 = fib(15)
    i = i + 1
```

Figure 4-1: The original Python source code used for benchmarking.

these benchmarks all use constructs that Starkiller's compiler cannot currently handle. The benchmark program used is shown in Figure 4-1. It consists entirely of calling the Fibonacci and factorial functions repeatedly, in a loop.

This benchmark is rather unfair in the sense that it exposes CPython's weakest points. For example, it relies heavily on calls of pure Python functions and does not take advantage of Python's fast native code implementations for manipulating large batches of data. As a result, it should not be considered representative of the performance gains one might expect from Starkiller once it is completed. Conversely, however, Starkiller is very immature in comparison to CPython, which has benefited from several years of extensive profiling and performance tuning. Almost none of the optimizations slated for inclusion in Starkiller's compiler have been implemented yet. These results are most relevant when understood as showing how the worst-case implementation of Starkiller compares against the most recent version of CPython on a benchmark that is as favorable to Starkiller as possible. In other words, as Starkiller matures, we expect to see better performance than shown below on a more representative sample of benchmark programs.

For comparison, Psyco and Python2C also participated in the benchmark. Due to a memory management bug, Python2C required human intervention in the form of a small change to the generated code in order to produce an executable that successfully completed the benchmark. Blitz was excluded from the test because it is such a special purpose tool and the problem domain it was designed for has no overlap with the limited functionality that Starkiller's compiler currently provides. Because it is not readily available, 211 was also excluded. As an additional point of reference, I included an equivalent program written by hand in C. The point of

Compiler	Lines of Code	Execution time	Relative Improvement
Python 2.3.3	–	173 seconds	1
Python2C 1.0a4	893	78 seconds	2.2
Psyco 1.2	–	7.1 seconds	24
Starkiller	65	2.8 seconds	62
Human generated C code	26	2.7 seconds	64

Table 4.1: Benchmark results comparison

this addition was to illustrate what the maximum possible performance was for this benchmark. All test programs were compiled using gcc 3.3.3 with -O3 optimization and no debugging or profiling support enabled; tests ran on a quiescent 800 MHz Pentium III with 256 MB of RAM. The results are as shown in Table 4.1. For each compilation system tested, this table shows the number of lines of code generated for intermediate compilation, the execution time, and the factor by which execution time was improved relative to the performance of Python 2.3.3. Execution time was measured by running all programs using the Unix `time` utility and examining the wall clock (real) time that had elapsed.

4.3 Analysis

As the results above indicate, Starkiller outperforms CPython by a substantial margin and Psyco by a smaller, but still significant margin. It is about 4% slower than human optimized code. The handmade C code is shown in Figure 4-2 while the code that Starkiller generated is shown in Figures 4-3 and 4-4.

As the generated source code indicates, Starkiller defined C++ classes to represent the main module (`mod0`) as well as the function definitions for `fact` (`fact0`) and `fib` (`fib1`). The actual definition of these functions at runtime is represented by the construction of a new instance of these functions' classes. If these function definitions included default arguments, the values of those arguments would be stored as instance variables in the function classes `fact0` and `fib1` after being passed to the class constructor. The application of both the functions and the module body itself is encapsulated in a method that overloads the function call syntax. The functions are referenced through pointer indirection, but this is not a major burden for the optimizer since the particular function to execute can be determined by the C++ type of the function. Within function and module bodies, variables are declared before use with their monomorphic type. It is not obvious from this example, but had there been polymorphic code present, Starkiller would have generated several overloaded function call methods for each polymorphic function class. These overloaded methods would differ in the types of arguments they accepted. Note that none of the methods Starkiller defines are virtual and neither C++ inheritance nor C++ run time type information is used.

Figures 4-5 and 4-6 show a portion of the C program generated by Python2C. This

```
#include <stdio.h>
int fact(int n) {
  if (0 != (n > 1)) {
    return 1;
  } else {
    return n*fact(n - 1);
  }
}

int fib(int n) {
  if (0 != (n < 3)) {
    return 1;
  } else {
    return fib(n - 1) + fib(n - 2);
  }
}

int main() {
  int i = 0;
  int q, q2;
  while (0 != (i < 100000)) {
    q = fact(12);
    q2 = fib(15);
    i = i+ 1;
  }
}
```

Figure 4-2: Hand made C code for performing the same task as the original Python source code in Figure 4-1.

part is the C function that corresponds to the definition of the factorial function. It is included to illustrate what static compilation for Python looks like in the absence of static type inference. The generated code is littered with calls to check objects' types before performing faster versions of operations (`PyInt_Check`), as well as tedious and cumbersome error checks, such as verifying that object constructors do not return NULL. In addition, note the pervasive use of reference counting as indicated by calls to the macros `Py_INCREF` and `Py_DECREF`. It is plain to see that even a sophisticated C compiler will have difficulty optimizing this code.

Close examination of the code suggests a problem with this comparison: Python2C generated C code that tested for integer overflow when performing arithmetic operations while the code generated by Starkiller did not. Psyco and the normal Python VM also checked for overflow while the hand made C program did not. For this particular benchmark program, none of the arithmetic exceeded what could be contained in a 32-bit signed integer, so no overflow occurred. Consequently, implementations that coerce integer objects to longs in the case of overflow were not penalized for

such conversions since they never took place. However, those implementations were penalized because they were continually performing checks that both the hand made C program and the code generated by Starkiller did not have to perform.

In reality, this discrepancy does not significantly change the results. Although it does not include such support right now, Starkiller will soon include the ability to check for integer overflow using the CPU's overflow trap as described in Section 3.2.1. In such cases, integer overflow triggers a CPU exception, which is caught by the operating system and eventually passed to the offending code by way of Unix signals. Recall that this benchmark does not trigger overflow at any time. Because this mechanism for overflow detection imposes zero overhead when no overflow occurs, its addition to the code generated by Starkiller is unlikely to change the results.

```
#include <iostream>
using namespace std;

class mod0 {
  public:
    class fact0;
    static fact0 *fact;
    class fib1;
    static fib1 *fib;

    class fact0 {
      public:
        int operator() (int n) {
            if (0 != (n <= 1)) {
                return 1;
            } else {
                return (n * (*fact) ((n - 1)));
            }
        }
    };

    class fib1 {
      public:
        int operator() (int n) {
            if (0 != (n < 3)) {
                return 1;
            } else {
                return ((*fib) ((n - 1)) + (*fib) ((n - 2)));
            }
        }
    };
```

Figure 4-3: The C++ code generated by Starkiller corresponding to the Python code in Figure 4-1, part 1 of 2.

```
    void operator() () {
        int i;
        int q;
        int q2;
        mod0::fact = new fact0();
        mod0::fib = new fib1();
        i = 0;
        while (0 != (i < 100000)) {
            q = (*fact) (12);
            q2 = (*fib) (15);
            i = (i + 1);
        }
    }
};

mod0::fact0 * mod0::fact = NULL;
mod0::fib1 * mod0::fib = NULL;

int main()
{
    mod0 m = mod0();
    m();
}
```

Figure 4-4: The C++ code generated by Starkiller corresponding to the Python code in Figure 4-1, part 2 of 2.

```
static PyObject *t_fact(PyObject * self, PyObject * args, PyObject * kw)
{
    int finally_code = FC_NORMAL;
    PyObject *result = NULL;
    PyObject *l_n = NULL;
    PyObject *temp_0 = NULL;
    PyObject *temp_1 = NULL;
    PyObject *temp_2 = NULL;
    PyObject *temp_3 = NULL;
    PyObject *temp_4 = NULL;
    PyObject *temp_5 = NULL;
    PyObject *temp_6 = NULL;
    PyObject *temp_7 = NULL;
    PyObject *temp_8 = NULL;
    long long_0;
    long long_1;
    long long_2;

    if (args == NULL)
        goto error;
    long_0 = ((PyTupleObject *) (args))->ob_size;
    if (!t_kwarg(args, "|O", kw, NULL, long_0, 1, t_fact_argnames, &l_n)) {

        return NULL;
    }
    Py_INCREF(l_n);

    do {
        long_0 = 1;
        if (PyInt_Check(l_n) && PyInt_Check(cInt_0)) {
            /* INLINE: cmp(int, int) */
            register long a, b;
            a = ((PyIntObject *) l_n)->ob_ival;
            b = ((PyIntObject *) cInt_0)->ob_ival;
            long_0 = a <= b;
        } else {
            long_0 = PyObject_Compare(l_n, cInt_0);
            long_0 = long_0 <= 0;
        }
    } while (0);

    if (long_0) {
        Py_XDECREF(l_n);
        return cInt_0;
    } else {
        temp_4 = PyTuple_New(1);
        if (temp_4 == NULL)
            goto error;
        long_2 = 1;
```

Figure 4-5: C code for the factorial function generated by Python2C for the program shown in Figure 4-1, part 1 of 2.

```
        if (PyInt_Check(l_n) && PyInt_Check(cInt_0)) {
            /* INLINE: int - int */
            register long a, b, i;
            a = ((PyIntObject *) l_n)->ob_ival;
            b = ((PyIntObject *) cInt_0)->ob_ival;
            i = a - b;
            if ((i ^ a) < 0 && (i ^ b) < 0) {
                PyErr_SetString(PyExc_OverflowError, "integer overflow");
                temp_6 = NULL;
            } else
                temp_6 = PyInt_FromLong(i);
        } else
            temp_6 = PyNumber_Subtract(l_n, cInt_0);
        if (temp_6 == NULL)
            goto error;
        PyTuple_SET_ITEM(temp_4, 0, temp_6);
        temp_6 = NULL;          /* Done with temp_6 */
        temp_3 = t_fact(NULL, temp_4, NULL);
        Py_DECREF(temp_4);
        temp_4 = NULL;          /* release called on temp_4 */
        if (temp_3 == NULL)
            goto error;
        if (PyInt_Check(l_n) && PyInt_Check(temp_3)) {
            /* INLINE: int * int */
            register long a, b, i;
            a = ((PyIntObject *) l_n)->ob_ival;
            b = ((PyIntObject *) temp_3)->ob_ival;
            i = a * b;
            if ((i ^ a) < 0 && (i ^ b) < 0) {
                PyErr_SetString(PyExc_OverflowError, "integer overflow");
                temp_0 = NULL;
            } else
                temp_0 = PyInt_FromLong(i);
        } else
            temp_0 = PyNumber_Multiply(l_n, temp_3);
        Py_DECREF(temp_3);
        temp_3 = NULL;          /* release called on temp_3 */
        if (temp_0 == NULL)
            goto error;
        Py_XDECREF(l_n);
        return temp_0;
    }
    Py_XDECREF(l_n);

    Py_INCREF(Py_None);
    return Py_None;
  error:
    Py_XDECREF(l_n);    Py_XDECREF(temp_0);    Py_XDECREF(temp_1);
    Py_XDECREF(temp_2);    Py_XDECREF(temp_3);    Py_XDECREF(temp_4);
    Py_XDECREF(temp_5);    Py_XDECREF(temp_6);    Py_XDECREF(temp_7);
    Py_XDECREF(temp_8);
    return NULL;
}
```

Figure 4-6: C code for the factorial function generated by Python2C for the program shown in Figure 4-1, part 2 of 2.

Chapter 5

Conclusions

5.1 Contributions

Starkiller is a research prototype of a type inferencer and compiler for the Python language. Although it is still immature, its design is complete and early tests show that it can provide substantial performance improvements through a combination of static compilation and static type inference. As demonstrated by previous compilation systems, static compilation for highly dynamic languages such as Python yields little if any benefit unless it is informed by a sophisticated static type inference algorithm. Although research compilers have been built for other dynamic languages, Starkiller is unique in offering better performance amongst interesting languages that are widespread enough to matter outside of academia.

The adaption of Agesen's Cartesian Product Algorithm to languages more widely deployed than Self constitutes a significant contribution. By elucidating the types of most expressions in a program at compile-time, Starkiller can discover the information needed to support many of the optimization algorithms that languages like Fortran and C have relied on for decades. Consequently, this work demonstrates that static type inference is more likely than originally thought to be feasible and beneficial, even for very dynamic languages with only latent types such as Python. Such languages expose terribly little information to compilers, making the compilers' job far more difficult.

Starkiller's type inferencer is far more than a simple port of Agesen's work from Self to Python. It introduces several new contributions such as:

Recursive customization Whereas Agesen relied on heuristics to detect and terminate recursive customization when it occurred during analysis, Starkiller avoids the issue entirely. It does so by representing out of scope references in a function as additional, "silent" arguments whose types participate fully in type inference. Since simple recursive call cycles involve a function calling itself, a simple recursive function will be among the nonlocal arguments over which its own templates are specialized. However, because Starkiller employs an online cycle detection algorithm that refuses to generate new types which contain

previously generated types originating from that node, template specialization terminates naturally.

Extension Type Description Language Starkiller's Extension Type Description language allows extension authors to participate fully in the type inference process. While many research systems exist that make use of programmer supplied annotations, the combination of writing annotations in the original source language along with the notion of supporting mixed language execution constitute a novel contribution. The former gives extension authors the ability to extend the type inferencer in a very powerful way; code written as part of an extension type description has full access to Starkiller's internals in addition to the Python runtime library.

Tainted containers The notion of tainted lists as external type descriptions demonstrates another significant contribution to the development of CPA–like algorithms. Such lists preserve maximal precision in the face of uncertainty while degrading gracefully as new information emerges that necessitates falling back to less precise behavior. More importantly, tainted lists do this all without ever violating the monotonicity invariant that CPA relies on.

Data polymorphism Agesen's original work had no support for data polymorphism, concentrating instead on parametric polymorphism. Starkiller's type inference algorithm handles data polymorphism reasonably well. Specifically, if two instances of a class have different types assigned to the same attribute, Starkiller will correctly infer the types for each without commingling them, ensuring maximal precision. This precision comes at the cost of analysis time and memory, however, empirical evidence suggests this is not overly significant and, in any event, that aspect of Starkiller's type inference algorithm is amenable to optimization. The core idea is that object types are duplicated at each creation site, so that objects created in different places (or different templates of the same function) are analyzed separately and do not interfere with one another.

Closures Starkiller handling of closures differs substantially from Agesen's, in large part because closures in Python cannot issue nonlocal returns. Whereas Agesen's algorithm tracks closure types using references back to templates, Starkiller's algorithm tracks closures by taking the cartesian product of the list of type sets they depend on. This approach is much simpler and and works without having to engage in the block fusion efforts that Agesen's algorithm must in order to work efficiently in the face of closures. The result is that in Starkiller's type inference algorithm, closure support is better integrated with CPA.

The notion of improving performance by excising interesting features from a language is (unfortunately) hardly novel, but Starkiller's restriction on dynamic code insertion is novel in other ways. Among dynamic languages, the Python community is unique in its insistence that language evolution be dictated by the demands of real world usage and experience rather than more abstract goals like amenability to analysis, conceptual unification, or, worst of all, elegance. In much the same spirit, the

decision not to support dynamic code insertion was made based on an analysis of how Python programmers in the real world work and what they need. That decision is justified by a compelling workflow narrative which shows that many developers who work in compute-intensive problem domains will benefit from using Starkiller, even if they lose access to the dynamic code insertion functionality that Python normally provides.

Beyond the type inferencer, Starkiller also includes contributions from its compiler. Most significant is the analysis of polymorphic function dispatch. Contributions in this area include the notion of reducing dynamic dispatch of polymorphic functions to simpler and faster single dispatch using signature hashing. Furthermore, the adaption of perfect hashing to fast polymorphic dispatch is another novel contribution. Both of these techniques are made possible by the elimination of dynamic code insertion. That restriction makes Starkiller rather unusual since there are no widely deployed languages that are both dynamic enough to need fast dynamic polymorphic function dispatch while at the same time being sufficiently static to ban dynamic code insertion.

Starkiller's most significant contribution may be its most mundane: the slow advancement of the state of the art made possible by changes in the world around us. In order to understand how Starkiller improves on existing technology, one must understand why its design outperforms CPython. One reason that Starkiller outperforms CPython is that the latter suffered from "Naive Implementor's Syndrome" initially leading to design defects that were compounded and crystallized by many years of incremental development [40]. A more significant reason is that Starkiller is a special purpose system and thus need not appeal to everyone; those that do not care for the design tradeoffs it represents can always return to CPython without losing anything. But the most significant difference is that the technology landscape in 2004 is vastly different from what it was in the early 1990s. There has been vast consolidations in markets for operating systems, compilers, and architectures. gcc now dominates; it is not as fast as the fastest compilers, but it is respectable and it is higher quality than most. More importantly, it is utterly pervasive, available on every architecture one can imagine. As a result, language implementors need not dumb down their implementation to the lowest common denominator since they can rely on the existence of the same high performance standards compliant compiler on every platform. Other compiler vendors such as Green Hills and Intel have been forced to implement gcc extensions. In a similar manner, GNU/Linux is everywhere. Proprietary Unix still lives, but looks increasingly like Linux as Solaris, AIX, and HP-UX rapidly adopt Linux compatibility layers. The same story unfolds in hardware. Since most architecture vendors have committed suicide, the scientific computing market has been ceded almost entirely to IA-32, x86-64, and PPC. For scientific computing, no other platforms matter statistically. No one outside of Intel believes that IA-64 is anything but dead on arrival. Alpha is dead. MIPS is dead. Cray is dead. PA-RISC is dead. Sparc is on life support since Sun canceled UltraSparc V and then UltraSparc VI development amid serious financial trouble.

5.2 Future Work

Despite the significant contributions this work represents, it has inspired more questions than answers, suggesting numerous lines of inquiry that future researchers might pursue. Before such research can begin however, construction of the current implementation must be completed. Starkiller must be, if not production ready, at least capable of analyzing and compiling the majority of Python programs available that do not use eval and its ilk. Completing Starkiller allows us to compare its performance on standard benchmarks like PyStone and Pybench, as well as analyze how well it performs on programs from different application domains. More importantly, getting Starkiller to the point where it understands all of Python enables us to determine how well Starkiller scales given large input programs. Until now, the answers have been purely conjectural.

Once Starkiller has matured sufficiently, it will provide an excellent substrate from which to experiment with dynamic language implementation techniques. Experimenting with different dynamic dispatch mechanisms on several different architectures is one example of useful research that may follow. The interactions between code size and branch prediction performance makes this problem particularly challenging and unlikely to be resolved by anything short of direct experimental observations. Several other pertinent areas of research are outlined below.

5.2.1 False Numeric Polymorphism

Consider again the basic polymorphic factorial program shown in Figure 2-4. Starkiller's analysis of this program will be flawed: it will determine that the return value of `factorial(5)` will be an integer but that the return value of `factorial(3.14)` will be either an integer or a floating point number. The problem here is that even given a floating point argument, factorial really will return the integer value 1 for some values of the argument `n`. This result is unintuitive because programmers expect that integer values are coerced into floats whenever the two types appear together in a binary operation. The fact that Starkiller will generate slower polymorphic code for this function is thus also not expected. An astute programmer can work around the issue by replacing the statement `return 1` with `return type(n)(1)` which coerces the constant one to the type of the argument `n`; unfortunately, Starkiller is not yet sophisticated to properly infer the result, although it easily could be in the future. Another workaround replaces the problematic return statement with `return 1 + (n - n)`; this workaround has the benefit of working with Starkiller right now while also being more likely to be optimized away.

Ideally, one would not have to resort to awkward workarounds in order to get inference results that match intuitive expectations. Indeed, in more elegant languages like Haskell, numbers are overloaded to prevent precisely this sort of problem. The crux of the problem here is that programmers' intuition does not match the semantics of the language; since Starkiller rigorously implements Python's semantics (at least in this case), confusion results. Future work is needed to uncover mechanisms for either automatically detecting problems like this or providing programmers with

better means of informing the compiler of their intentions.

5.2.2 Static Error Detection

The standard Python implementation provides a consistent model for handling errors. A small class of errors, mostly syntactic in nature, is detected by the byte compiler when a program is compiled or run for the first time. These errors typically abort compilation or execution entirely. All other errors are detected at runtime and are dealt with using Python's exception handling mechanism. In practice, this means that many errors that are detected statically in other languages, such as calling a function with more arguments than are expected, become run time errors that are only detected upon execution. This error handling model poses a problem for Starkiller: what should one do with potential errors detected statically by the type inferencer?

```
x = "hi there"
x = 4
y = x + 3
```

Figure 5-1: An example of static detection of run time errors.

To illustrate the problem, consider the simple program in Figure 5-1. This code will compile and execute correctly in the standard Python implementation without issue. But when Starkiller analyzes this code, it will conclude that the variable `x` can have a type of either string or integer. Since `x` is then used as an operand for an addition with an integer, Starkiller cannot blindly invoke the addition without resolving the `x`'s polymorphism. Because it is flow insensitive (see Section 2.1), Starkiller cannot determine that the string value of `x` will not be present by the time control reaches the addition expression. Having failed to statically resolve `x`'s polymorphism, we must do so dynamically by inserting a run time type check on `x` before proceeding with the addition. If `x` is an integer, then the addition can proceed without harm. But if `x` is a string, we have a problem, since the addition operator is not defined for operand types of string and integer.

What should the type inferencer due at this point? Aborting compilation because an error is possible would be unwise in this case since we know that the string type associated with `x` is a spurious artifact of the type inference algorithm and not something that will hinder the addition operation. At the same time, we cannot simply ignore the issue since it could truly be a programmer error. Consider the same case as before, but with the first two lines swapped. That program would compile correctly but would generate a runtime exception when fed into the standard Python implementation. Python's inability to statically detect errors like this as well as simpler errors such as typographic errors in variable names has been a major complaint from its user base since its inception. That suggests that Starkiller should make static errors visible to the user so they can check and repair incorrect code. Unfortunately, doing so would also produce spurious warnings as seen in the first example. More-

over, due to the effects of templating, the same error may be detected many times, drowning the user in a sea of meaningless errors with no easy way to separate the spurious from the legitimate.

The core of the problem is that some of the errors that Starkiller statically detects represent real errors that the standard Python implementation cannot statically detect while others represent artifacts of the type inferencer's imprecision and are completely harmless. There is no way, a priori, to discriminate between the two. Because Starkiller focuses on improving run time performance and not necessarily run time safety, it foregoes reporting statically detected errors such as those described above to the user. However, it does not ignore possible error cases. Instead, it inserts code to raise run time exceptions as needed when it statically detects that an error is possible and dynamically determines that an error has occurred. This approach preserves the standard Python semantics while bypassing the need to build an extensive error reporting and filtering system. Nevertheless, future research would be well directed at improving static error detection and presentation to developers.

5.2.3 Integer Promotion

Python offers two builtin integer datatypes: a standard "machine" integer that must be at least 32-bits wide and a long integer that can have unbounded length. Traditionally, these two datatypes were kept distinct and isolated from one another. This meant that when a machine integer value overflowed, the Python Virtual Machine would raise an `OverflowError` exception. Long integers, of course, cannot overflow since they use arbitrary precision arithmetic to grow as needed. Starkiller's problem stems from the fact that recent Python releases have adopted a change in semantics [51] designed to eventually unify the machine and long integer types. Newer Python releases now respond to overflow of a machine integer by silently promoting it into a long integer. The eventual goal of these changes is to eliminate machine integers completely, but that is not expected to happen for quite some time.

While unifying the two integer datatypes does solve some very real problems in the language, it introduces a new problems for Starkiller, namely how one efficiently implements silent coercion semantics without ruining performance in the common case where overflow does not occur. Note that this problem is orthogonal to the issue of detecting overflow at runtime as described in Section 3.2.1. Regardless of which version of Python's semantics Starkiller implements, it must detect overflow. The question to be addressed here is what to do once integer overflow is detected. Agesen faced precisely this same problem when working on type inference for Self [1], but the only solution he suggested was offering users an option to treat integer overflow as a hard error rather than silently coercing to BigInts. This is the same choice that Starkiller makes.

There are four options for Starkiller to implement Python's newer integer semantics. The first approach requires that Starkiller mirror the language definition precisely. The second option would be to implement sophisticated techniques for range analysis and propagation associated with previous work in using unboxed integers in higher level languages. The third option would be to uniformly use long integers

everywhere, while the fourth option would be to retain the integer semantics found in older versions of Python. At the moment, Starkiller uses the fourth option, opting for high performance and limited implementation complexity at the cost of conformance with the language specification.

Precisely mirroring the language definition means that machine and long integers remain distinct types but that all operations that could overflow a machine integer must be checked at runtime for overflow and promoted to long integers as needed. From a type inference perspective, this means that all primitive integer operations such as addition and multiplication may return either machine integers or long integers. As a result, almost all integer variables will be polymorphic since they must hold machine integers as well as long integers. Making all integer variables polymorphic cripples the performance of generated code by inhibiting the use of unboxed arithmetic. The standard Python Virtual Machine does not face this problem since it uniformly uses boxed integers anyway, and, in any case, it has enough overhead to mask that caused by integer promotions.

The literature associated with compiler optimization for dynamic languages is full of techniques for partially reclaiming the performance lost to integer boxing [16, 46]. Unfortunately, many of these strategies necessitate a degree of implementation sophistication that is not presently available to Starkiller. Recent work on unboxing in statically typed languages [27] such as Haskell and the ML family of languages may prove more appropriate to Starkiller's static compilation model while posing less of an implementation hazard.

In contrast, using long integers exclusively offers somewhat better performance than is possible with strict conformance to the standard since all integer variables can be at least monomorphic. Long integers could even be implemented in an unboxed manner, if they were implemented as a linked list of machine words. In that implementation, long integers would contain one word of the integer data and a pointer to the succeeding word. For most integers, the next pointer would be NULL, signifying that this was the last word comprising the integer. As a result, long integers would be twice the size of machine integers in the common case and could be stack allocated but would incur extra overhead needed to constantly check if the next pointer was null. This overhead would manifest itself in repeated branch points inserted into the instruction stream for code that was heavily laden with arithmetic operations. Excess branching hinders performance by confounding processor pipelining in modern architectures [20]. Some of this overhead could probably be ameliorated by judiciously marking such branches using the GCC extension function `__builtin_expect`, which directs the compiler to insert a hint to the target processor's branch prediction unit that one end of the branch is unlikely to be taken. The primary problem with this approach is that while it does represent the direction in which the Python language is moving towards, Python is not there yet, so it would represent a divergence from the official language definition.

A related solution involves boxing integers directly, where all integers are represented by a machine word that contains either a pointer to a long integer, or a 31-bit integer. This representation uses one bit of the integer to determine whether the object is a machine integer or a pointer. While well accepted in the Smalltalk com-

munity, this approach has been harshly criticized in the Python community, making its introduction into Starkiller problematic. Much of the criticism has centered on difficulties achieving efficient implementations across many different processor architectures; the Alpha architecture in particular imposes a severe performance penalty on the bit twiddling operations required to support this approach. Many of the other concerns raised center around implementation complexity and would not be relevant to Starkiller.

An even greater divergence from the language specification is implied by the third option, namely keeping the original Python semantics of isolating machine from long integers. Machine integer overflow would result in an exception rather than silent promotion to long. This option offers the best performance while introducing significant semantic differences compared to the Python language specification.

Regardless of how integer overflow is handled, performance can be improved by not using overflow recovery mechanisms wherever static analysis determines that integer overflow is impossible. Fortunately, there are a number of powerful algorithms that perform just such an analysis, such as those described in [19, 49, 17, 26]. These algorithms determine the integer ranges which variables and expressions can take during the program's lifetime. Static knowledge of variable ranges allows one to easily see that some arithmetic operations cannot overflow under any circumstances; these operations can be safely performed using unsafe arithmetic operations with no provision for handling overflow. Range analysis enables another optimization, bounds check elimination. Indeed, bounds check elimination is often the primary motivation for range analysis research since languages like Java mandate bounds checked array access but lack automatic coercion to long integers in response to overflow.

The bad news is that none of the range analysis algorithms surveyed mesh particularly well with Starkiller's type inference algorithm. The good news, however, is that many of them are amenable to application on the typed intermediate language that Starkiller's type inferencer produces as output. The better news is that, when it comes to range analysis, a little goes a long way. Kolte and Wolfe report in [26] that performing the simplest of range analysis algorithms removes 98% of bounds checks in a program while the more sophisticated algorithms yielded only minor incremental improvements. Since Python's builtin containers can only have lengths that are represented in a machine word, that result suggests that such simple algorithms should also eliminate many overflow checks.

The problem we face then is how to integrate type inference with range analysis given that the two are mutually interdependent problems. Range analysis should come after type inference since without type inference, we cannot know what variables represent containers and what variables represent integers. However, in the absence of range analysis, type inference must conservatively assume that all arithmetic operations on integers can overflow and thus return long integers as well as machine integers. Even if a later range analysis pass deduces that many of these arithmetic operations cannot overflow, it is too late to undo the effects of the overflowed long integers once they have propagated into the constraint network. CPA relies entirely on the monotonicity of the type sets, so removing types after they have been inserted is impossible.

When Agesen was faced with a similar challenge involving two mutually interdependent problems, he was clever and created CPA [1]. Lacking cleverness, at least for the moment, I opt instead for a less efficient but simpler brute force solution. The algorithm that results is a three pass effort in which type inference is followed by range analysis followed by another type inference round. The initial type inference round makes pessimistic assumptions that all integer operations can overflow and generate longs. When the range analysis pass runs, it notes which variables the previous pass marked as having integer (either machine or long) type and which variables were marked as containers. The resulting range analysis is used to mark some container accesses as safe, enabling the compiler to omit bounds checks. In addition, arithmetic operations that range analysis indicates cannot overflow are marked specially. On the second type inference pass, integer arithmetic operations that are marked as being incapable of overflow generate only integer result types. This algorithm easily generalizes to multiple rounds, but given the nature of the information passed between rounds, there seems no additional benefit to performing additional range analysis passes. Note that tuples and untainted lists benefit most from the bounds check elimination since their length is statically known. Integrating the bounds check elimination into the External Type Description Language in a more general way so that authors of extension containers (such as Numeric Python or NumArray) can benefit by eliminating unneeded bounds checks for provably safe container accesses remains an open problem.

5.2.4 Eval, Exec, and Dynamic Module Loading

The most glaring deficiency in Starkiller's type inference algorithm is its inability to handle dynamic code generation. Starkiller needs to be able to see all the code that could be executed at runtime in order to perform complete inference. As a result, constructs such as `eval` and `exec` which evaluate and execute source code potentially made available only at run time are problematic. For the same reason, Python's module system can theoretically be used to import code that does not exist at compile time since `import` statements are evaluated only at run time.

While Python offers several methods to execute code generated at runtime, in practice, `eval`, `exec`, and module imports that cannot be resolves statically are rarely used. In part, this is because dynamic code generation introduces security risks for the same reason that it makes static type inference difficult: code that has not been audited and verified prior to distribution cannot be trusted, especially if that code could have been contaminated by untrusted user data. Another reason is that code generated dynamically often runs slower than comparable static code. This slowdown is due to the fact that the dynamically generated code must be byte compiled before execution, while static code is byte compiled only once on application startup and then cached for successive executions.

Finally, dynamic code generation is infrequently used in the Python world because Python makes it easy to accomplish the same ends as achieved with dynamic code generation using other means. For example, in order to convert the string `"1"` into the corresponding integer, one could use the `eval` function or one could

simply use the `int` function. The latter has the advantage that its return value will always be an integer should it return at all. In addition, the developer can rest secure in the knowledge that no extraneous code can be invoked by a call to `int`, regardless of the value of the input string. As another example, consider the case of a developer attempting to read an attribute from an object `instance` where the attribute name is derived from run time data. Solutions to this problem using `eval` such as `eval(‘‘instance.’’ + userInput())` compare poorly in security, performance, and aesthetics when compared to more traditional solutions such as `getattr(instance, userInput())` or even `instance.__dict__[userInput()]`. This is especially true when one considers the fact that the return value of `userInput()` could include parenthesis, triggering a method call when evaluated.

To determine how serious a problem Starkiller's inability to support dynamic code generation may be, I performed a survey of the library code written in Python that was shipped with release 2.3 of Python. I examined all instances where `eval` and `exec` where used and recorded the difficulty involved in working around their use. The results are shown in the following table. One immediate observation is that `eval` and `exec` are infrequently used: there were less than 25 occurrences of both functions in over 123,000 lines of source code. Moreover, many of those occurrences can be easily removed in favor of alternative facilities afforded by the language. All of the remaining uses occur in programs or libraries that are simply not suited for Starkiller's target audience. These programs include heavily interactive applications like the Python Debugger and the interface to libreadline in addition to more esoteric applications, like the compiler infrastructure. The foregoing analysis is meant to be illustrative rather than authoritative; dynamic code insertion is more prevalent than the prevalence of `eval` alone would suggest. Alternative forms such as dynamic class loading and some applications of serialization are arguably more prevalent but also more difficult to measure.

`bdb.py`, `pdb.py`
Debugger support
Not applicable for Starkiller. Implements run time support for the Python debugger, and uses `eval` to modify variables at run time after stopping at a breakpoint.

`dumbdbm.py`
Dumb DBM Clone
Trivial to work around. Uses `eval` to cheaply parse string into three numbers. Security hazard since that allows arbitrary code execution.

`gettext.py`
Internationalization and Localization support
Trivial to work around using named functions.

`gopherlib.py`
Gopher protocol support
Easy to work around using dictionaries instead of module variables as a static mapping, or, alternatively, a call to `globals`.

`mhlib.py`
Unix Mail Handling
Not applicable to Starkiller since `eval` is only used in testing code.

`os.py`
Generic Operating System interface
Moderately difficult to work around. `eval` is used to support conditional compilation. `eval` could be eliminated if `os.py` was better modularized with autoconf, but this is not for the faint of heart. Alternatively, `globals` could be used.

`random.py`
Random number generator
Not applicable to Starkiller since `eval` is only used in testing code.

`reconvert.py`
Conversion between old and new regular expressions
Not applicable to Starkiller since this code is only used during development.

`rexec.py`
Deprecated restricted execution framework
Very difficult to work around, but this code is going to be removed soon since it is terminally insecure.

`rlcompleter.py`
Word completion for libreadline
Not applicable to Starkiller since this code is only used in interactive applications.

`tzparse.py`
Experimental parser for timezone specifications
Trivial to work around. Uses `eval` to parse simple strings into numbers.

`warnings.py`
Generic warning framework
Moderately difficult to work around. Uses `eval` to check if user supplied names are defined. Could probably be replaced with dictionary mapping or a much safer call to `getattr`.

`compiler/ transformer.py`
Experimental Python parser
Moderately difficult to work around. Uses `eval` to parse literals in a way that precisely matches the C implementation of Python's parser. `eval` could probably be replaced if a more detailed literal parser was written in Python.

`logging/ config.py`
Configuration parser for logging system
Moderately difficult to work around. Uses `eval` to map class name strings to the corresponding class objects. `eval` could probably be replaced by an explicit dictionary mapping or a call to `getattr`. This is also a potential security vulnerability allowing arbitrary code execution.

`idlelib/ CallTrips.py`, `ObjectBrowser.py`, `PyShell.py`
Integrated Development Environment
Not applicable to Starkiller since this code uses `eval` to support dynamic introspection and live object modification for an IDE.

Even if dynamic code insertion is not used very often in production programs, its loss may still be sharply felt if many developers utilize a workflow that depends on it. For example, one of Python's most touted benefits is the ability to develop code interactively, modifying code in an editor while periodically reloading the code into a running interpreter. This approach speeds development by offering the developer immediate real–time feedback to source code changes. Python's introspection and reflection features coupled with its lack of object access controls give developers the ability to interrogate and modify all objects at run time, which only further increases the benefits of interactive development. Moreover, this style of interactive development can proceed much faster if the ability to simply reload and test new code allows developers to bypass time consuming initialization functionality. This is particularly relevant in cases where development requires establishing a connection to a database in which starting the connection might be extremely slow while retrieving data on an existing connection is fast. Another example in which interactive development proceeds faster is the case of scientific computing programs that perform simple manipulations on large datasets. Due to the growing disparities between CPU and IO performance, reading the data from disk can take much longer than actually manipulating it. In such cases, development can be much faster if the dataset can be retained in memory without needing to be reread from disk every time a code modification is made and tested.

Despite the productivity benefits, Starkiller's current inability to support interactive development is unlikely to prove a major hindrance to its adoption for several reasons. For example, not all application domains and development philosophies benefit from this workflow. Arguably, adoption of test–first development methodologies greatly diminishes the benefits associated with interactive development. In addition,

many developers highly value the simplicity and reliability that come from certainty in knowing that the system started in a known state with no stale objects lingering from previous runs. This is especially true when developers use the interpreter to make changes in the running system so that the state of the program in the editor does not reflect the state of the system. Moreover, Starkiller can support a limited form of interactive development, namely the ability to inspect objects interactively in an application shell, for example, after an exception has occurred. Such inspection has no impact on type inference and requires far less support than supporting dynamic code insertion in general.

Finally, lack of interactive development is not a major impediment to Starkiller's adoption because there already exists an excellent interactive development environment that supports the Python language: it is called CPython. Developers can prototype their applications in CPython with small datasets for testing and then move to Starkiller when the simple bugs have been worked out and they are ready to work on large datasets. This gives them the best of both worlds, and is especially appropriate for scientific computing applications where the analysis of a large dataset can take hours or days to complete. Scientific computing environments have already adopted a similar workflow because they develop programs on an individual desktop but eventually test them on large compute clusters where debugging is more difficult and interactive development is impractical if not altogether impossible.

Between the current workflow model in which all interactive development is foisted onto CPython and full blown support of dynamic code insertion lies an intermediate option. Starkiller could offer the option of compiling to modules to C++ library files that support CPython's extension API. In effect, Starkiller would utilize the same workflow as Python2C originally intended: develop in CPython until things get too slow, then compile performance critical modules into extensions and continue developing in CPython. The resulting code could not interact completely freely with arbitrary Python code since Starkiller could not have analyzed how arbitrary code interacted with compiled modules. Consequently, such modules operate under restrictions. For example, classes defined in compiled modules would be sealed to prevent Python code outside the extension from simply replacing methods with new function blocks that Starkiller has not seen, cannot analyze, and may have already inlined in other parts of the module. The result is something like extension types in CPython right now: one cannot reassign the methods on the builtin type int even though one can subclass int. This intermediate approach brings with it a raft of new challenges that must be addressed in future research. One significant primary challenge is how to integrate CPython objects with Starkiller objects. If Starkiller adopts CPython's object library and representation, it may lose a significant performance benefit. Alternative approaches require judicious conversions whenever objects cross the boundary layer between CPython and Starkiller code.

Despite the problems inherent in supporting dynamic code insertion in a high performance static compilation environment, Starkiller will eventually support some (perhaps limited) form of dynamic code insertion. We describe possible implementation strategies presently. The most immediate problem faced in implementing eval is the need for a parser and interpreter or compiler at run time. Once the dynamic code

has been parsed and interpretation begins, other problems quickly emerge. Consider the following possibilities:

1. the evaluated code could call functions with argument types that do not match any existing template

2. the evaluated code could assign a value with a type not seen before as an object's attribute

3. the evaluated code could replace code or data that had previously been assumed to be constant and thus inlined

The first problem is easily solvable if we mandate that the compiler generate a generic template for each function and method in addition to whatever specialized templates the type inferencer indicates are required. Such generalized templates treat all arguments as polymorphic and perform explicit type checks throughout in much the same manner as the current implementation of Python does today. Including them imposes no performance overhead beyond increased executable size. Starkiller needs to be able to generate such polymorphic templates when compiling megamorphic functions in order to avoid combinatorial explosion. The second and third problems are interrelated and pose far more difficult challenges. In and of itself, the second problem is not insurmountable since any data member could easily be replaced with a pointer to new tagged data. The larger difficulty stems from the existence of (possibly inlined) code throughout the system that accesses that data.

There are two possible solutions to this problem. One solution would involve placing layers of indirection so that any code or data that could be modified by dynamically inserted code could be easily replaced. This would impose a severe performance penalty unless developers were given some method to declare code "sealed" from further changes. The second solution requires that Starkiller recompile the entire world image while running whenever it needs to handle dynamically inserted code. The overhead involved in this recompilation would be substantial, making it unsuitable for frequent uses of the `eval` function and practical for little more than updating modules in the field. However, this recompilation process is amenable to several optimizations. For example, recompilation might only be used as a last resort if the evaluation interpreter ran into any of the three problems described above. In the common case, evaluation would proceed without incident. Once recompilation was triggered, the resulting world image would be generalized in ways that would make recompilation less likely to be needed again. Another optimization requires that Starkiller track dependencies between different functions, methods, and data definitions so that it can recompile only the parts of the program that are affected by the evaluated code. However, this approach presents further performance challenges since it requires that Starkiller take full control of inlining away from the target compiler. Moreover, it necessitates either keeping most of the type inferencer's internal state from compile time accessible at run time in order to handle new code or running the type inferencer over the entire program for each new change.

In general, the need to perform run time code patching is difficult to reconcile with the peak performance since very high performance compilation techniques make

heavy use of inlining and other techniques that destroy program modularity. The conflict between high performance and dynamic code generation is not impossible to reconcile, but it remains quite challenging.

Bibliography

[1] Ole Agesen. *Concrete Type Inference: Delivering Object-Oriented Applications*. PhD thesis, Stanford University, 1996.

[2] John Aycock. Converting Python Virtual Machine Code to C. In *The Seventh International Python Conference*, November 1998. Available at `http://www.foretec.com/python/workshops/1998-11/proceedings/papers/ayco%ck-211/aycock211.html`.

[3] Jonathan Bachrach and Glenn Burke. Partial Dispatch: Optimizing Dynamically–Dispatched Multimethod Calls with Compile–Time Types and Runtime Feedback, 1999. Available at `http://www.ai.mit.edu/~jrb/Projects/pd.pdf`.

[4] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[5] H. Boehm and M. Weiser. Garbage Collection in an Uncooperative Environment. *Software Practice and Experience*, pages 807–820, September 1988. Also see `http://www.hpl.hp.com/personal/Hans_Boehm/gc/`.

[6] Frederick Brooks. *The Mythical Man Month*. Addison Wesley, 1995.

[7] Luca Cardelli. Basic Polymorphic Typechecking. *Science of Computer Programming*, 8(2):147–172, 1987.

[8] Jong-Deok Choi, Manish Gupta, Mauricio J. Serrano, Vugranam C. Sreedhar, and Samuel P. Midkiff. Escape Analysis for Java. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 1–19, 1999.

[9] Stephen Cleary. Boost Pool Library, 2001. Available at `http://www.boost.org/libs/pool/doc/`.

[10] Tom DeMarco and Timothy Lister. *Peopleware: Productive Projects and Teams*. Dorset House Publishing, 1999.

[11] Abrahams et al. Technical Report on C++ Performance. Technical report, ISO International Standardization Working Group for the C++ Programming

Langauge, August 2003. Available at http://anubis.dkuug.dk/jtc1/sc22/wg21/docs/PDTR18015.pdf.

[12] D.G. Bobrow et al. Common LISP Object System Specification X3J13 Document 88-002R. *ACM SIGPLAN Notices*, 23, Sep 1988.

[13] Ittai Anati et al. The Intel Pentium M Processor: Microarchitecture and Performance. *Intel Technology Journal*, 7(2), May 2003.

[14] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements od Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY, 1995.

[15] David Gay and Bjarne Steensgaard. Fast Escape Analysis and Stack Allocation for Object-Based Programs. In *th International Conference on Compiler Construction (CC'2000)*, volume 1781. Springer-Verlag, 2000.

[16] Jean Goubault. Generalized Boxings, Congruences and Partial Inlining.

[17] John Gough and Herbert Klaeren. Eliminating Range Checks Using Static Single Assignment Form. In *Proceedings of the 19th Australasian Computer Conference*, 1996.

[18] Ralph E. Griswold and Madge T. Griswold. *The Icon Programming Language*. Prentice Hall, 1990.

[19] Rajiv Gupta. Optimizing Array Bound Checks Using Flow Analysis. *ACM Letters on Programming Languages and Systems*, 2(1-4):135–150, March–December 1993.

[20] John L. Hennessey and David A. Patterson. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, 1998.

[21] Jim Hugunin. Python and Java: The Best of Both Worlds. In *6th International Python Conference*. Corporation for National Research Initiatives, October 1997. Available at http://www.python.org/workshops/1997-10/proceedings/hugunin.html.

[22] Jim Hugunin. IronPython: A fast Python implementation for .NET and Mono. In *PyCon*. Python Software Foundation, March 2004. Available at http://www.python.org/pycon/dc2004/papers/9/IronPython_PyCon2004.html.

[23] Bob Jenkins. Minimal Perfect Hashing, 1999. Available at http://burtleburtle.net/bob/hash/perfect.html.

[24] Eric Jones. *Weave User's Guide*, 2003. Available at http://www.scipy.org/documentation/weave/weaveusersguide.html.

[25] Eric Kidd. Efficient Compression of Generic Function Dispatch Tables. Technical Report TR2001-404, Dartmouth College, Computer Science, Hanover, NH, June 2001.

[26] Priyadarshan Kolte and Michael Wolfe. Elimination of Redundant Array Subscript Range Checks. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 270–278. ACM Press, 1995.

[27] Xavier Leroy. The effectiveness of type-based unboxing. In *Workshop Types in Compilation '97*. Technical report BCCS-97-03, Boston College, Computer Science Department, June 1997.

[28] David Mertz. Charming Python: Make Python run as fast as C with Psyco, October 2002. Available at `http://www-106.ibm.com/developerworks/linux/library/l-psyco.html`.

[29] Stephan Murer, Stephen Omohundro, David Stoutamire, and Clemens Szyperski. Iteration abstraction in Sather. *ACM Trans. Program. Lang. Syst.*, 18(1):1–15, 1996.

[30] Lutz Prechelt. An Empirical Comparison of Seven Programming Languages. Technical Report 2000-5, University of Karlsruhe, March 2000.

[31] Armin Rigo. *Representation-based Just-in-time Specialization and the Psyco prototype for Python*, 2004. Available at `http://psyco.sourceforge.net/theory_psyco.pdf`.

[32] Neil Schemenauer, Tim Peters, and Magnus Lie Hetland. Simple Generators. Python Enhancement Proposal 255, May 2001.

[33] Bruce Schneier. Computer Security: Will We Ever Learn? *Cryptogram*, May 2000.

[34] Bruce Schneier. Phone Hacking: The Next Generation. *Cryptogram*, July 2001.

[35] Andrew Shalit, David Moon, and Orca Starbuck. *The Dylan Reference Manual: The Definitive Guide to the New Object-Oriented Dynamic Language*. Addison-Wesley, 1996.

[36] Stephen Shankland and Ben Charny. Linux to power most Motorola phones, February 2003. Available at `http://news.com.com/2100-1001-984424.html`.

[37] Olin Shivers. The semantics of Scheme control-flow analysis. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, volume 26, pages 190–198, New Haven, CN, June 1991.

[38] Y.N. Srikant and P Shankar, editors. *The Compiler Design Handbook: Optimizations and Machine Code Generation*. CRC Press, 2003.

[39] Bruce Sterling. *The Hacker Crackdown: Law and Disorder on the Electronic Frontier*. Bantam Books, 1993.

[40] Dan Sugalski. Performance problems in perl, python, and parrot. Private communication on April 5 and May 16–19, 2004.

[41] Allison Taylor. AT&T to invest $3 billion in 2003 for global network, September 2003. Available at `http://www.infoworld.com/article/03/09/11/HNattnetwork_1.html?source=rs%s`.

[42] Computer Emergency Response Team. CERT Advisory CA-2001-19 "Code Red" Worm Exploiting Buffer Overflow In IIS Indexing Service DLL, 2001.

[43] Computer Emergency Response Team. CERT Advisory CA-2001-26 Nimda Worm, 2001.

[44] Computer Emergency Response Team. CERT Advisory CA-2003-04 MS-SQL Server Worm, 2003.

[45] Computer Emergency Response Team. CERT Advisory CA-2003-16 Buffer Overflow in Microsoft RPC, 2003.

[46] Peter J. Thiemann. Unboxed values and polymorphic typing revisited. In *Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 24–35. ACM Press, 1995.

[47] Guido van Rossum. Subtyping Built-in Types. Python Enhancement Proposal 252, May 2001.

[48] Guido van Rossum and Fracis L. Drake, editors. *Python Language Reference*. PythonLabs, 2003.

[49] Hongwei Xi and Songtao Xia. Towards Array Bound Check Elimination in Java Virtual Machine Language. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 14. IBM Press, 1999.

[50] Ka-Ping Yee and Guido van Rossum. Iterators. Python Enhancement Proposal 234, January 2001.

[51] Moshe Zadka and Guido van Rossum. Unifying Long Integers and Integers. Python Enhancement Proposal 237, March 2001.

[52] Olivier Zendra, Dominique Colnet, and Suzanne Collin. Efficient Dynamic Dispatch without Virtual Function Tables: The SmallEiffel Compiler. pages 125–141.