Static Type Inference and Compilation with Starkiller

Michael Salib

May 24, 2004

Abstract

Pure Python code is slow, primarily due to the dynamic nature of the language. I have begun building a compiler named Starkiller to produce fast native code from Python source programs. While compilation can improve performance, any such gains will be modest unless the compiler can statically resolve most of the dynamism present in source programs. Static type inference for Python programs would enable the safe removal of most type checks and most instances of dynamic dispatch and dynamic binding from the generated code. Removing dynamic dispatch and binding leads to large performance benefits since their existence precludes many traditional optimization techniques, such as inlining. Starkiller includes a static type inferencer. Given a Python source file, it can deduce the types of all expressions in the program without actually running it. This paper is describes Starkiller's design and operation.

Contents

1 Overview

Pure Python code is slow. Most of the time, that does not matter because application performance generally does not depend on pure Python very much. Nevertheless, there are situations where it does matter, so I have spent time trying to discover how to improve the performance of pure Python code. I believe that static compilation is the answer.

The greatest obstacle to efficient compilation and high performance is Python's dynamic nature: every operation must verify that its operands are of the correct type, every access requires going through layers of indirection, etc. The combination of dynamic binding and dynamic typing is particularly problematic for optimization since it postpones all dispatch decisions until runtime, thereby eliminating most benefits of static compilation. This dynamism makes programming in Python a joy, but generating optimal code a nightmare. Yet while the presence of such abundant dynamism makes traditional static optimization impossible, in most programs, there is surprisingly little dynamism present. For example, in most Python programs:

1. all class and function objects are defined exactly once
2. class inheritance relationships do not change at run time
3. methods are not added after a class object has been created
4. most expressions have exactly one type; the vast majority of those that have more than one type have only a few types

The flip side is that what little dynamism a particular program makes use of is often absolutely vital.

I have developed a type inference algorithm for Python that is able to resolve most dispatches statically. This algorithm is based on Ole Agesen's Cartesian Product Algorithm for the type inference of Self programs. I have built a type inferencer for Python based on this algorithm and incorporated it into the Starkiller compiler. Starkiller's type inference algorithm works by constructing a dataflow network for types that models the runtime behavior of values in an input program. 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 modelled 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 have at least the elements of the sender node's type set. In other words, types flow along constraints.

To avoid both loss of precision from merging types in function bodies and an exponential growth in the analysis time required, Starkiller takes the cartesian product of the list of argument type sets seen by any function call. The result is a list of argument type lists in which each argument list contains exactly one type for each argument. Starkiller performs analysis on function bodies for each of these monomorphic argument lists, caching the results so they can be used at other call sites. This ensures that polymorphic function calls do not pollute the analysis of functions called later in the call chain.

In much the same way, Starkiller deals with class instances by maintaining polymorphic state describing the instance's attribute types at the instance construction site. Taking the cartesian product of this state allows us to generate monomorphic instance states that flow from a instance definition node. These monomorphic instance states are immutable; when they encounter a `setattr` instruction, they do not change their own state, but pass the request back to their polymorphic parent who then changes its state and generates new monomorphic instance state types as appropriate.

The key idea behind Starkiller is that it never removes or retracts type information. At any point in time, an expression's node's type set accurately, but perhaps incompletely, describes the types that expression will observe at run time. This monotonicity ensures that Starkiller will eventually converge on a final type state for the entire program. The last piece of the puzzle is understanding how external types interact with the type system. Starkiller provides a mini-language that allows extension type authors to describe the type behavior of their extensions. These descriptions are used during inference to model the run time dataflow. Since most extensions are quite simple from a type perspective, these descriptions tend to be short and simple. For example, the `str` type constructor takes any object and returns a string instance (possibly calling the object's `__str__` method if it exists).

Like Agesen [1], we necessarily take an operational approach in describing Starkiller's type inference algorithm. We thus forego mathematically intensive descriptions of the denotational semantics in favor of simpler step-by-step descriptions of what the algorithm does. To ease exposition, we begin by describing Starkiller's type inference algorithm for the most basic language features: assignments, function definitions and calls, and the object system. We then examine the 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 type inference algorithm works for pure Python code, we then turn our attention to Starkiller's support for foreign code 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 Basic Type Inference 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. A concrete type is a type that are actually instantiated at runtime, as opposed to an abstract type that is either never instantiated (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 analysis one module at a time.

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 4) can be readily adapted.

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 type inference 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. Techniques that may one day allow the integration of these features into Starkiller are discussed briefly in [11].

2.1 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 1: Assignment in action.

As an example, consider the source code in Figure 1 and the corresponding constraint network shown in Figure 2. 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 sets 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 that represent 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.

2.2 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 3 serves to create a function object and bind it to the name "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 4. 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.

Upon encountering a function definition, Starkiller creates a special function definition node that encapsulates the code associated with the definition. It also creates a variable 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 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

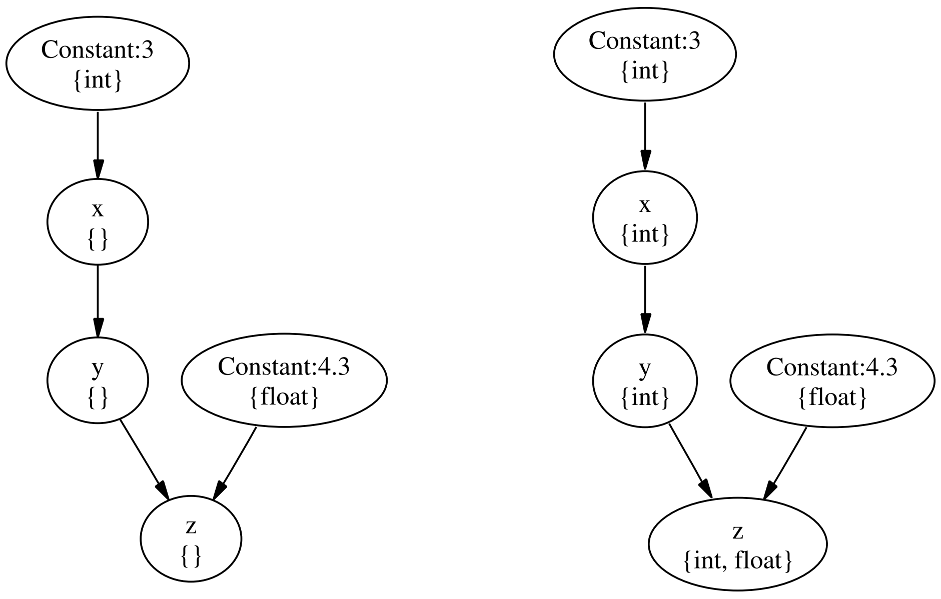

Figure 2: Constraint network for the code shown in Figure 1 before (left) and after (right) type propagation has converged.

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

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

Figure 3: The factorial function.

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,

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

Figure 4: A nested function.

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 4, 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 as there are in Agesen's system [1].

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 3. 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. 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 provides significant improvements over the heuristics Agesen employed to detect and stop recursive customization.

2.3 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 and also calls the class' constructor method as a side effect, passing in the newly formed instance object and the constructor call arguments. 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 writes 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 lists of monomorphic types are packaged into monomorphic 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.

3 Advanced Language Features

3.1 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 define 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 [14].

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.

3.2 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.

3.3 Iterators and Generators

Recent Python releases have introduced iterators [16] and generators [12] into the language. Their design was heavily influenced by the Sather [10] and Icon [5] 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.

3.4 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.

4 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.

4.1 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`.

4.2 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 5. This version does not implement the "tainted" behavior 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

```
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 5: An external type description for the list type from the builtins module

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.

5 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.

5.1 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.

5.2 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.

This problem will be made clearer with an example, shown in Figure 6. 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

```
class a:
    pass

def f():
    return a()

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

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

Figure 6: Template instance shadowing in action.

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.

5.3 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 7 which portrays a stereotypical implementation of the Visitor design pattern [2] 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 7. 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.

```
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 7: The visitor design pattern in Python.

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.

5.4 False Numeric Polymorphism

Consider again the basic polymorphic factorial program shown in Figure 3. 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.

5.5 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 8: An example of static detection of run time errors.

To illustrate the problem, consider the simple program in Figure 8. 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), 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. Moreover, 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.

5.6 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 [17] 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 **??**. 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 [3, 13]. 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 [9] 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 [7]. 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 community, 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 [6, 15, 4, 8]. 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 [8] 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.

References

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

[2] 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.

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

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

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

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

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

[8] 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.

[9] 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.

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

[11] Michael Salib. Starkiller: A Static Type Inferencer and Compiler for Python. Master's thesis, Massachusetts Institute of Technology, 2004.

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

[13] 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.

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

[15] 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.

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

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