Shed Skin

An Optimizing Python-to-C++ Compiler

Master's Thesis, Mark Dufour

April 19, 2006

mark.dufour@gmail.com
Parallel and Distributed Systems Group, Delft University of Technology

Shed Skin

An Optimizing Python-to-C++ Compiler

Master's Thesis, Mark Dufour

Committee:

Dr.ir. K.G. Langendoen
Prof.dr.ir. H.J. Sips
Dr.ing. L.M.F. Moonen

Acknowledgements

This document marks the end of a long journey. My heartfelt gratitude goes out to those that inspired me to set out on this journey, and to those that have kept me from giving up.

Thank you Richard Stallman, Guido van Rossum, Ken Thompson, Dennis Ritchie and Bjarne Stroustrup for inspiration, and for the awesome, marketing-free software you have created.

Thank you Leonardo *****, for helping me fix many bugs, for many good ideas, and for always sticking with me, even though I did not always have the energy to respond to your mails.

Thank you Google, for sponsoring my work, as part of the Summer of Code project. Thank you also, for showing that it is possible to compete in todays world, by making good products.

Thank you Marijn Heule, for showing me what real brilliance is like, for a great software project (March), and for many interesting experiences - landing me in Italy and my favourite television show, for example!

Thank you Koen Langendoen, for taking on this crazy project, even though it was just a crazy idea at first, and for diligently guiding me through the formalities.

And finally, thank you Haifang, for your relaxed attitude, your support throughout the process of writing this thesis, and for keeping up with my obsessive nature.

Abstract

We investigate the possibility of converting implicitly, but statically typed Python programs into efficient C++ code. The main challenge is to perform static type inference. This is difficult, because polymorphism causes types to be context-dependent. The secondary challenge is to transform inefficient heap allocation, if possible, into stack- and static preallocation.

We describe an experimental Python-to-C++ compiler, called Shed Skin, that accepts implicitly, but statically typed Python programs. To precisely and efficiently perform static type inference for parametric and data polymorphic code, it combines the Cartesian product algorithm with iterative class splitting. To transform heap allocation into stack- and static preallocation, it uses a simple invariant-based escape analysis.

We show that our compiler precisely and efficiently analyzes a set of 16 non-trivial benchmarks, and that the analysis time is mostly related to the 'degree' of polymorphism, rather than program size. We show that the run-time performance of the generated code for our benchmarks typically outperforms that of Psyco, the current state-of-the-art in optimizing Python programs, by a factor of 2-40.

Contents

1 Introduction **1**

2 Related Work **3**

2.1 Static Type Inference 3

2.1.1 Parametric Polymorphism 5

2.1.2 Data Polymorphism 6

2.2 Stack- and Static Preallocation 8

2.2.1 Stack Allocation 8

2.2.2 Static Preallocation 9

3 Shed Skin **10**

3.1 Static Type Inference 10

3.2 Stack- and Static Preallocation 12

4 Implementation **13**

4.1 Static Type Inference 13

4.1.1 Iterative Class Splitting 14

4.2 Code Generation 19

4.3 Python Features 21

4.3.1 Language Features 21

4.3.2 Builtin Types, Methods 24

4.3.3 Standard Library 29

5 Evaluation **30**

5.1 Analysis Time 31

5.2 Run-time Performance 34

6 Conclusion **36**

7 Future Work **38**

7.1 Scalability 38

7.2 Precision 39

7.3 Optimization 40

7.4 Integration 41

References **42**

A Othello Player (Python) **44**

B Othello Player (C++) **46**

1 Introduction

'Premature optimization is the root of all evil.'
C.A.R. Hoare

Mainstream imperative languages have traditionally been designed with much consideration for *efficiency*. But as computers are becoming ever more powerful, efficiency is becoming less of an issue in an increasing number of situations. This means efficiency can increasingly be traded for programmer *productivity*, without noticeable impact on performance.

Java was the first mainstream imperative language to purposefully sacrifice much efficiency to increase productivity. It essentially did so by providing a simplified version of C++. From a language perspective, it improved productivity over C++, by adopting a *uniform reference model* and automating memory management.

More than ten years after the initial release of Java, greatly improved computer speeds allow us to go even further. New imperative languages have appeared, designed from the ground up to maximize productivity. The most popular of these languages, Python [18], essentially adds a complete *run-time model* to Java, along with *high-level syntax* and *data types*.

While becoming less of an issue, efficiency will always be important in some cases, such as scientific computing, and welcome in many other cases. However, efficiency that is traded for productivity is not necessarily lost: while Java was initially very slow, advances in compiler technology have made it much more efficient.

In this thesis, we investigate compiler technology to improve the execution speed of Python programs. We have already mentioned the two major sources of inefficiency:

1. A complete run-time model allows any binding to be changed dynamically, so that all method calls are, in principle, dynamically dispatched.

2. A uniform reference model means that all objects are, in principle, individually allocated and garbage collected on the heap.

The run-time model is the most critical of the two: if we cannot optimize basic method calls, no amount of effort will make Python an efficient language. However, while most programs benefit from the high-level syntax and data types, many programs are not very dynamic at all. In fact, they can often be expressed, without major changes, in a completely *static* language such as C++.

Therefore, an interesting direction of research is to combine the high-level syntax of Python with the static typing of C++. One of the major advantages of a run-time model is that types are *implicit*. By using implicit, but *static* typing, we can maintain this advantage. Moreover, this might allow us to generate efficient C++ code, provided we can automatically generate type declarations.

This leads to the following question, which we try to answer in this thesis:

- *Is it possible to convert implicitly, but statically typed Python programs into efficient C++ code?*

A compiler that can perform this task must use two types of techniques, corresponding to the mentioned inefficiencies. First, to generate *type declarations*, it must be able to determine the type of each program expression. Thus, it needs to employ *static type inference* techniques. Second, to avoid heap allocation if possible, it must use techniques to transform heap allocation into *stack-* and *static preallocation*.

To investigate these techniques, we have developed a simple Python-to-C++ compiler, called Shed Skin. Its aim is to convert *unmodified*, but statically typed Python programs into efficient C++ code. It is explicitly not meant to compile arbitrary Python programs. As such, we believe it can be a useful niche compiler for use in scientific computing.

2 Related Work

While dynamic, imperative languages have a long history, powerful static type inference techniques have appeared only recently. In part, this is because there are other, much simpler, optimization approaches, such as *type feedback* [10], that improve performance for arbitrary programs. Moreover, it is still unclear, if static type inference can be made scalable enough to analyze large programs.

Techniques to perform stack- and static preallocation have also appeared only recently. In part, this is because they require a *static call graph* to be effective (in other words, static type information). Because a run-time and uniform reference model usually go together, researchers have only turned to such techniques after the emergence of Java.

For a broader overview of existing techniques to optimize dynamic languages, and their historical development, the reader is referred to our prior literature research [6].

2.1 Static Type Inference

To statically bind, or otherwise optimize, dynamic method dispatches we need to estimate the types of receiver expressions. Estimates can be obtained via a combination of *profiling* and *inference*. Profiling provides us with a *lower bound* on possible types, whereas type inference provides us with an *upper bound*. Lower bounds can be used to optimize method calls, as they predict the types of receiver expressions. Upper bounds are typically more useful, as they facilitate static binding. However, they are also much harder to obtain.

Overview Static type inference is difficult, because we cannot deduce a single type for each syntactic expression: *polymorphism* causes types to be *context-dependent*. An analysis that does not separate different contexts causes these types to be mixed, resulting in massive imprecision. The following example illustrates the problem:

```
a = max(1, 2)
b = max(2.0, 3.0)
```

Example 1: The types inside `max` are context-dependent. An analysis that mixes these types is imprecise for both `a` and `b`.

Fortunately, two common types of polymorphism are amenable to analysis. First, as shown above, formal argument types often depend on actual call sites. This type of polymorphism is called *parametric polymorphism.* Second, instance variable types often depend on actual allocation sites. This type of polymorphism is called *data polymorphism.* The following example shows both types of polymorphism:

```
a = max(1, 2)
b = max(2.0, 3.0)

c = [1, 2]
d = [2.0, 3.0]
```

Example 2: For parametric polymorphism, types depend on actual call sites; for data polymorphism, types depend on actual allocation sites.

To separate contexts during type inference, we need to *duplicate* code. This gives rise to two problems. First, we do not know which contexts need duplication, since this is dependent on actual types, which is the thing we are trying to determine. Second, contexts may be *nested*: consider, for example, an allocation site within a function whose arguments depend on context, inside another function, etc.

An intuitive approach to separate contexts is to use *syntax-based duplication*: preemptively duplicate functions for each call site, and classes for each allocation site. However, besides redundant, this approach is also imprecise, since it does not consider nested contexts. To analyze nested contexts, we can preemptively duplicate code for nested contexts, up to a certain depth. However, this causes an exponential growth in code size [1], and leaves us with a method that is still imprecise for arbitrarily deep polymorphism.

To be practical, type inference methods need to *adapt* to the actual polymorphism used, that is, to the actual types they are trying to determine. This can be done in two ways. The first approach is to use *partial type information* available during analysis, to direct duplication decisions. However, since partial type information is by definition incomplete, this appears to necessarily lead to imperfect decisions. But decisions cannot be easily retracted, since the partial solution set has to be be monotonically growing [1]. The second approach is to *iterate* the analysis, allowing for easy retraction of decisions between iterations.

2.1.1 Parametric Polymorphism

Agesen's *Cartesian Product Algorithm* [2] is an arbitrarily precise, adaptive, single-pass technique to analyze *parametrically polymorphic* code. It exploits partial type information in such a way that decisions never have to be retracted.

During the analysis, functions are duplicated for each element of the *Cartesian product* of partial argument types. Each element corresponds to a possible combination of actual arguments during run-time. Since elements are by definition *monomorphic*, there is no imprecision and decisions never have to be retracted. New elements can simply be added as partial type information grows. The following figure illustrates the idea:

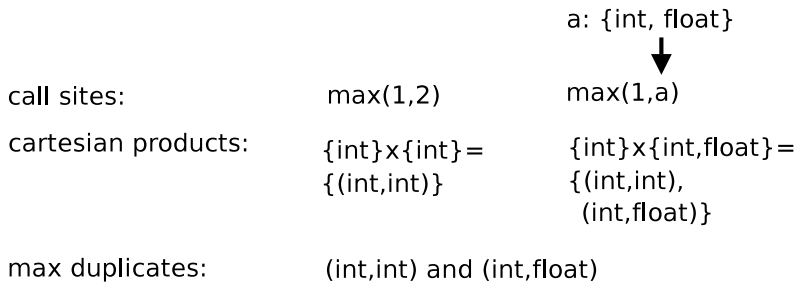

Figure 1: The Cartesian product algorithm duplicates functions for each element of the Cartesian product of partial argument types.

The Cartesian product algorithm is more efficient than iterative solutions, because it avoids any imprecision. In contrast, the first iteration of an iterative solution is by definition very imprecise, since duplication only happens afterwards. This typically makes the first iteration very slow, since the analysis may keep propagating types, and many subsequent iterations may be required to recover from the initial imprecision.

A disadvantage of the Cartesian product algorithm is that the size of Cartesian products is *exponential* in the number of arguments. Fortunately, this number is generally very limited, and argument types consist of at most all class types.

2.1.2 Data Polymorphism

The behaviour of simple data types such as scalars never changes, so all allocation sites can be given a definite type. The behaviour of *data polymorphic* types such as `list` depends on how their instance variables are used within the context of an allocation site. However, this usage is fundamentally unknown at the moment an allocation site is seeded during the analysis. But the decision of whether to share or duplicate a class has to be made here. The following example illustrates the problem:

```
a = []
b = []
ident(a).append(1)
ident(b).append(1.0)
```

Example 3: To determine the types of the two allocation sites, we need to duplicate `ident`, but this in turn depends on these types.

Wang and Smith [20] have developed a *single-pass* extension to the Cartesian product algoritm to adaptively analyze data polymorphic code. However, their analysis appears to be very complicated, as well as dependent on heuristics for its precision. Furthermore, it is unclear if it will work for dynamic languages, as the authors have only published results for Java.

A much simpler, *iterative* solution was developed by Plevyak, as part of the first analysis to adaptively analyze both parametric and data polymorphic code, called *iterative flow analysis* [15]. It works by iteratively splitting classes, based on observed *imprecisions*. Imprecisions, as in the above example, are caused by conflicting assignments to a single instance variable.

Starting at conflicting assignments, a backward data-flow analysis is performed, tracking the involved allocation sites for each assignment. A *confluence point* is a node in the data-flow graph, through which allocation sites flow to different assignments. If there exists a confluence point, the respective class is split for allocation sites that flow along different incoming data-flow edges. The class is split directly for allocation sites that do not flow to a confluence point.

The following figure illustrates the process, for Example 3 above:

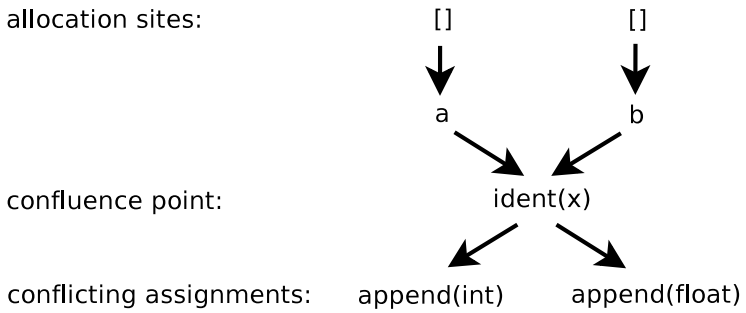

Figure 2: The `list` class is split along the two incoming data-flow edges of the confluence point, so that during the next iteration, the two allocation sites are seeded with different class duplicates.

2.2 Stack- and Static Preallocation

By reasoning about *object lifetimes*, compilers can perform many, traditionally manual, memory optimizations. Most importantly, heap allocation can in many cases be transformed into *stack-* and *static preallocation*. Other promising optimizations are *object inlining* and *combining* [5, 19], that decrease the overhead of heap allocation by combining multiple objects.

2.2.1 Stack Allocation

Stack allocation can be performed, when it is clear that an object does not outlive the stack frame of the method in which it is allocated. Otherwise, the object is said to *escape*. An *escape analysis* conservatively approximates escapement of local variables. Stack allocation can be performed for allocation sites that are not assigned to an escaping variable.

Choi et al. [4] have developed an escape analysis for Java. So-called *connection graphs* are created for each method, capturing the relationships between allocated objects and local variables. An interprocedural reachability analysis is used to determine escapement of each variable. Choi et al. show that their technique identifies 20% of all objects as non-escaping, so they can be stack allocated, for ten large benchmarks.

Franz et al. [7] have developed a much simpler escape analysis, that is usually as precise as the approach of Choi et al. It is based on the observation, that if the following three invariants hold for a variable, it is guaranteed to be non-escaping:

1. It is not returned as a result.
2. It is only assigned to other non-escaping variables.
3. It is only passed to methods as a non-escaping formal.

The actual analysis is performed as follows. All global and instance variables are marked as escaping, the others as non-escaping. The program is scanned to see if it violates any of the invariants. If so, the corresponding variable is marked as escaping, and the process is repeated, until no further violations occur.

2.2.2 Static Preallocation

Gheorgioiu et al. [8] have recently developed an elegant technique to perform *static preallocation* for Java. Two allocation sites are called *compatible*, if no two objects allocated at the different sites can exist at the same time. A *unitary* allocation site is compatible with itself. For unitary sites, allocation can be performed statically. For compatible, unitary sites, preallocated memory can even be shared.

An object liveness analysis is used to conservatively estimate the set of objects that is live at each program point. As for the described escape analyses, only local variables are tracked. The result is used to determine pairs of incompatible allocation sites. Finally, a minimal graph coloring algorithm is applied. Sites that end up with the same color are guaranteed to be compatible.

Gheorgioiu et al. show that their technique identifies 60% of all allocation sites as unitary, so they can be statically preallocated, for twenty large benchmarks. Moreover, they show that sharing space for compatible, unitary sites reduces preallocated space by 95%.

3 Shed Skin

'Everything should be made as simple as possible, but no simpler.'
Albert Einstein

We have developed an experimental Python-to-C++ compiler, called Shed Skin, that combines several of the techniques described in the previous section. For its type inference part, the Cartesian product algorithm is combined with iterative class splitting. The simple invariant-based escape analysis is used to perform stack- and static preallocation.

We have chosen C++ as the target language, because it provides many reusable abstractions. Most importantly, its *generics* and flexible object-orientation provide us with support for *compile-time polymorphism*. It also allows for explicit stack- and static preallocation. Furthermore, Python features such as namespaces, exceptions and operator overloading can be mapped to their C++ equivalents.

Since in C++, variables must have an *abstract* or *generic type*, we do not support variable types that go beyond this. For example, a variable that can be either of type A or B at the same time, and that cannot be determined to be data, parametric or oo-polymorphic will be rejected by our compiler. Likewise, the contents of generic containers such as `list` must have an abstract or generic type. This way, types never have to be identified during run-time, and an *unboxing analysis* [12] is not necessary.

3.1 Static Type Inference

The Cartesian product algorithm is an elegant technique to adaptively analyze parametric polymorphism. However, it loses much of its precision for programs using data polymorphism: just consider Python's `list` type, which is typically used throughout programs.

We have described two solutions to this problem. Wang and Smith's extension to the Cartesian product algorithm appears overly complicated and depends on heuristics for its precision. While *iterative flow analysis* does not suffer from these flaws, it does not utilize partial type information in handling parametric polymorphism.

As suggested by Agesen [1], we have combined the Cartesian product algorithm with the data iterative part of iterative flow analysis. Each iteration consists of performing the Cartesian product algorithm. Class splitting is performed between iterations. In subsequent iterations, we consider duplicates of a single class as different types. This way, when class duplicates flow to a single function argument, the Cartesian product algorithm generates separate function duplicates.

The result is a conceptually simple technique that adaptively analyzes parametric and data polymorphic code, exploits partial type information and is independent of heuristics for its precision.

Recursive Customization We have not dealt with potential *recursive customization*. This occurs when recursion in the data-flow, often caused by imprecise intermediate results, causes function or class duplicates to be spawned indefinitely.

For example, consider an allocation site inside a duplicated function. Class splitting may cause duplicated allocation sites to be seeded with different types. If the allocation site now somehow flows back into an argument of the function, the allocation site is duplicated again, potentially causing further class splitting, etc.

While techniques exist to detect and deal with recursive customization [1], we have only observed a single case of non-termination. However, this may well have been caused by a bug in the compiler.

We believe there are two reasons why recursive customization may only occur sporadically. First, the Cartesian product algorithm greatly improves the precision of intermediate results over a fully iterative solution, avoiding much data-flow recursion. Second, being adaptive, our analysis focuses on resolving actually observed imprecisions, which avoids much useless class splitting.

Scalability The *scalability* of our analysis depends in large part on the first iteration. Since class splitting is only performed afterwards, data-flow recursion can cause the Cartesian product algorithm to generate many useless function duplicates. Even though the number of duplicates

is exponential in a generally very limited number of arguments, argument types may be very imprecise, when there are many classes. As a result, the first iteration typically makes up much of the complete analysis time. With improved precision, subsequent iterations are generally much more efficient.

To speed up the first iteration, we can place initial limits on the number of duplicates that can be generated for each function, and merge all function duplicates, for functions with many arguments. Instead, we have used a *heuristic* to provide the analysis with an initial 'guesstimate' of the type of each syntactical `list` allocation site. Heuristics have no effect on precision, since a fully adaptive analysis can always recover from wrong guesses.

To speed up subsequent iterations, we additionally *merge* class duplicates, when their instance variables all have the same, proven type. The requirement is crucial, since without it, we might end up splitting merged types again, potentially causing non-termination.

3.2 Stack- and Static Preallocation

We have used the invariant-based escape analysis, as described in Section 2.2.1, to transform heap allocation, if possible, into stack allocation. Based on escapement information, we additionally perform a static preallocation analysis. An allocation site can be statically preallocated, if it flows into a variable for which the following two invariants hold:

1. It is non-escaping.
2. Its function is not part of a call graph cycle.

A return value can be statically preallocated, when it is not assigned to an escaping variable, and the calling functions are not part of a call graph cycle. To avoid interference, we allocate separate space for each syntactic call.

4 Implementation

'There's always one more bug.'
Lubarsky's Law

Building a compiler that works for many programs requires a lot of effort. While conceptually simple, actually implementing iterative class splitting is less straightforward than it appears. Code generation is complicated by the need to deduce *abstract* and *generic* types. Finally, since Python is a relatively large language, writing a compiler for it requires much attention to details.

4.1 Static Type Inference

We have used Palsberg and Schwartzbach's *constaint-based approach* to model program flow [14]. For a given program, it entails deriving a set of *subtyping constraint*s. For example, an assignment $x := y$ is modeled by a constraint of the form: $type(x) \supseteq type(y)$. This is equivalent to saying that data 'flows' from y to x.

Dynamic method calls are modeled by *conditional constraints* between actual and formal arguments. For a method call $x.y(..)$, constraints of the following form are added, for each class C that implements a method named y: $C \in type(x) \Rightarrow type(formal) \supseteq type(actual)$.

The resulting constraint graph is solved by using a *fixed-point data-flow analysis* [13]: types are propagated along the constraint graph, until no further changes occur.

Conditional constraints cannot be used in a fully adaptive analysis, since it is unknown in advance, which class duplicates there will be in the end, or which method duplicate will be called for a given class type and method name. Therefore, borrowing from the Cartesian product algorithm, we *dynamically* add constraints, when the argument types at a call site grow. Depending on the types, we connect the call site to existing or new method duplicates.

Duplication of classes and methods is *two-dimensional*, since classes can contain methods. Therefore, nodes in our constraint graph are identified by a combination of the respective node in the abstract syntax tree, and an index for each dimension.

4.1.1 Iterative Class Splitting

Iterative class splitting works by observing and resolving imprecisions, caused by conflicting assignments to a single instance variable. Before discussing our implementation, we must clarify what is meant by 'conflicting assignment'. For our purposes, an assignment corresponds to an arbitrary set of types, flowing into an instance variable, along a single data-flow edge. Conflicting assignments correspond to different type sets.

In Python, instance variables are created dynamically. Because of intermediate imprecisions, we may encounter conflicting assignments to actually non-existent instance variables, leading to potentially useless splitting. Fortunately, we know the set of instance variables of builtin types. By initially focusing on these types, our analysis avoids splitting on non-existent variables. As builtin container types such as `list` are typically also the most polymorphic, splitting them first further resolves the most imprecision.

Classes with multiple instance variables can be handled by focusing on one variable at a time. Our analysis simply iterates over all classes, starting with the builtin types, and considers one instance variable at a time. Whenever a class is split, a new iteration is started. When no more imprecisions are found, all types have been precisely determined.

Splitting Process When we have located conflicting assignments to an instance variable, we determine the involved allocation sites for each set of equal assignments, by using a backward data-flow analysis. For example, for a statement $x.a := b$, the possible allocation sites of x are associated with an assignment to the current type of b.

If there are allocation sites that flow to only one type of assignment, we directly split the respective class. Otherwise, we use the results of the

backward analysis to determine a confluence point, or a node in the constraint graph, through which multiple allocation sites flow to conflicting assignments. We then split the class, according to the configuration of allocation sites flowing along the incoming data-flow edges of the confluence point.

Unfortunately, it is not always clear how to best split a class, since a single allocation site may flow along multiple edges. For example, consider two data-flow edges with associated allocation sites $\{a, b, c\}$ and $\{b, c\}$, respectively: since b and c always occur together, there is no reason to split them. In this case, the best split is into $\{a\}$, and $\{b, c\}$.

We have developed a simple splitting heuristic that tries to keep allocation sites such as b and c together, while trying to spread allocation sites over the edges. For each pair of edges, it simply removes the allocation sites, that occur along both edges, from the edge with the most allocation sites. The following algorithm literally shows our heuristic:

```
for (i, seti) in enumerate(edge_sets):
    for setj in edge_sets[i+1:]:
        in_both = seti.intersection(setj)

        if len(seti) > len(setj):
            seti -= in_both
        else:
            setj -= in_both
```

Algorithm 1: For each pair of edges, we remove the allocation sites that occur along both edges, from the edge with the most allocation sites.

In some cases, our heuristic is unable to determine a split: if each allocation site flows along each edge, it groups all of them together along a single edge. When this happens, we currently split the respective class for each of these allocation sites. In the future, we should probably consider multiple confluence points, before resorting to such drastic measures. This would also allow us to compare the usefulness of different confluence points, with regard to performing splitting.

Allocation Table Information about the type of each (duplicated) allocation site needs to be somehow maintained between iterations. One approach is to update the constraint graph after each iteration, removing parts that become unreachable. This can be done by cleverly relying on garbage collection [15].

Instead, we use a global *allocation table* to describe the current type of each allocation site. Entries are identified by a an allocation node in the abstract syntax tree, and an element of the Cartesian product for the respective parent method. The allocation table is regenerated during each iteration, based on actually duplicated methods. This way, irrelevant entries are cleaned out automatically.

Class splitting essentially creates new types. Therefore, methods, and hence allocation sites, may be duplicated for a combination of argument types not described by our allocation table. To seed these allocation sites with a type, we initially let them *share* the types of corresponding allocation sites in a '*mother*' method duplicate, that is guaranteed to be described. If necessary, iterative class splitting can split these types later on.

To identify 'mother' method duplicates, we maintain a list of which class splits have been performed since the previous iteration. This allows us to determine, for each argument type of a new method duplicate, if it has just been split, and if so, from which class duplicate. These duplicates, together with the types that were not split, all existed during the previous iteration, so a method duplicate for these types is guaranteed to be described. The following figure illustrates the process:

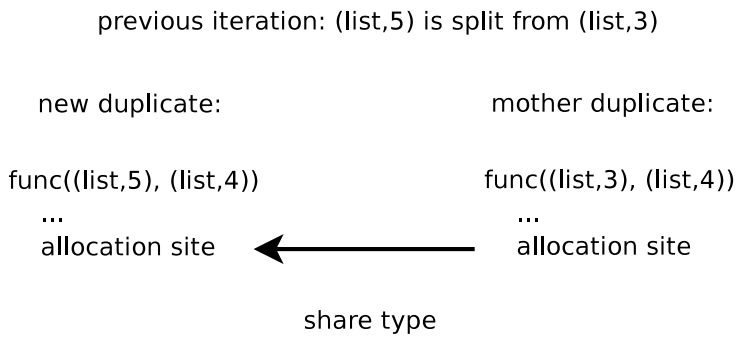

Figure 3: When a new method duplicate is not described by our allocation table, we identify a 'mother' duplicate that is guaranteed to be described, and we let corresponding allocation sites initially share their types.

Scalability Our heuristic to provide an initial 'guesstimate' of the type of each syntactical allocation site currently only considers lists, as these are typically the most polymorphic. It simply examines expressions determining the initial content of each syntactically allocated list, and groups together lists with the same type of content.

To determine the type of an expression, our heuristic only looks at the expression itself. It identifies three common cases: the expression is a (compound) constant, a nested allocation site for a builtin type, or a call site to a builtin function with a fixed type (currently, (`x`)`range`, and conversion functions such as `int`). Our heuristic recurses for nested list allocation sites, as these are very common.

As mentioned, we perform class merging after each iteration. A class duplicate can be safely merged, when its type is *proven*, i.e. we are sure it can never be split later on. This is the case when each instance variable is proven to be of a certain scalar type, or of another proven type. For example, lists of lists that are proven to always contain integers can be safely merged.

Incomplete Instances So far, we have ignored class instances that do not flow to an assignment, for certain instance variables. For example, an empty list may be assigned to a variable, without ever containing anything itself:

```
a = []
a = [1,2]
```

Example 4: The empty list does not flow to any instance variable assignment. However, it does flow together with an integer list.

As shown, such 'incomplete' instances generally do flow together with other instances of the same class, that do flow to assignments. Based on these other instances, we could assign a *static type* to them. Instead, we have chosen to appropriately *cast* them, since this allows for a limited form of run-time polymorphism.

To know when to use a cast, incomplete instances need to be distinguished during the analysis. To this end, we identify allocation sites that do not flow to an assignment during the first iteration, and associate them with a separate class duplicate. We have not generalized this process for classes with multiple instance variables.

4.2 Code Generation

With inferred types in hand, generating C++ code still presents a challenge. A major problem is having to deal with three types of (interacting) polymorphism. We also need to perform run-time conversions, to support mixing of types containing integers and floats.

Generic Type Declarations Because of parametric polymorphism, type declarations can depend on argument types; because of data polymorphism, they can depend on instance variable types. When generating type declarations, we need to consider both types of context.

Prior to actual code generation, we scan argument and instance variable types, and assign C++ template variables to polymorphic (sub)types. To generate a type declaration for a variable or expression, we recursively visit its type, while comparing each subtype with the found template variables within its context. If a subtype is equal to that of a template variable, for each respective (method or class) duplicate, we use the name of the template variable in place of the subtype.

Object-Oriented Polymorphism Without considering generics, generating type declarations for abstract (sub)types is easy: if a type can be one of several classes, we simply use the name of the lowest common ancestor. Converting Python class hierarchies to C++ forms a larger problem. Because of Python's run-time model, *virtual* method and variable declarations are not necessary. Therefore, in a similar vein as for type declarations, we need to infer these ourselves.

There are two situations in which we need to add virtual declarations. First, a method or variable declaration may be overridden. Second, a declaration that is accessed via a certain class, may only exist in its descendants. Of course, there may already be a virtual declaration with the same name in an ancestor class. In this case, we generate only the most common declaration.

Abstract and generic types may interact in subtle ways. For example, consider a generic class that is inherited from, or a list of objects of

type A and B, with a lowest common ancestor of type C: depending on context, it can be a generic list of type A or B, a list of abstract type C, or a generic list of type A, B or abstract type C. We currently do not support such mixing of generic and abstract types.

Run-time Conversions The run-time polymorphism we support by casting incomplete instances can be generalized to complete, 'compatible' instances. For example, consider a list of integers that flows together with a list of floats. Preemptively changing this list into a list of floats can have serious consequences on program behaviour. Therefore, since it appears useful to support this type of polymorphism, we have decided to convert compatible types during run-time.

We generate a conversion function, for each type of compatible assignments. Since types may be arbitrarily nested, these functions can call each other to convert subtypes.

The situation is complicated by assignments involving generic subtypes. Consider adding a list of integers to a list of floats: the formal argument to the addition operator of `list` has a generic list type. To determine if conversion is necessary for assignment to generic formal arguments, we look at the type of the receiver expression. In our example, this is a list of floats, so we decide to convert it.

We have only implemented run-time conversion for lists. Generalizing it to handle other types appears straightforward. Additionally, we have not yet incorporated casting into this more general approach.

4.3 Python Features

The main consideration in Python's evolving design is to maximize productivity. To be compact, yet readable, it necessarily supports many high-level features. Besides a run-time and uniform reference model, Python supports features such as *list comprehensions*, *high-level assignments* and *iteration*. It further contains high-level *builtin types* such as `list`, `tuple`, `dict`, `file` and `set`, as well as *builtin methods* such as `range`, `sum`, `max`, `zip` and `enumerate`. Finally, a very large *standard library* is available to perform almost any common task.

4.3.1 Language Features

When mapping Python features to C++, the evaluation order of expressions must remain the same, or side-effects may cause subtle differences in program behaviour. In general, we make sure expressions are appropriately evaluated, by using temporary variables and intermediate (template) functions. As an example of the latter, the expression `2*[1]` is mapped to `__mul(2, new list<int>(1))`, where `__mul(int n, T t)` is a template function that returns `t->__mul__(n)`.

List Comprehensions A list comprehension succinctly expresses a new list in terms of one or more 'iterable' objects. For example, `[2*x for x in range(10)]` creates a list of even numbers 0 to 18. As they are very common, list comprehensions greatly add to the polymorphic usage of the list class.

For each syntactical list comprehension, we generate a `static inline` function that accepts iterable objects and builds the respective list. Since expressions in a list comprehension are evaluated within the scope of its parent method, we additionally pass any needed, non-global identifiers as arguments.

List comprehensions are often nested. For example, `[[(x+y)%2 for x in range(8)] for y in range(8)]` creates a checkerboard pattern. This means we may have to pass identifiers through multiple generated func-

tions. List comprehensions can also be polymorphically overloaded, as part of a generic class or method. In this case, we generate corresponding template functions.

High-Level Assignments Python assignments can be grouped together using nested tuples, e.g. `a,b = b,a` and `a,(b,c) = b,(1,2)`. Variables are automatically *packed* and *unpacked,* when they are paired with tuples. For example, for `a,(b,c) = (x,y),z`, the tuple `(x,y)` is assigned to `a`, and the elements of the binary sequence `z` (e.g. a list, tuple or string of length 2) are assigned to `b` and `c`.

As actual arguments are assigned to formal arguments, and return values may be assigned to variables, Python also supports (un)packing in these cases, e.g. `x,y = f(z,(a,b))`, where `f` is defined as `f((u,v),w): return w`. Unpacking is further supported when iterating objects as part of a for loop (possibly inside a list comprehension), e.g. `[x+y for (x,y) in [(1,4),(4,9)]]`.

We support all of the preceding, except unpacking of actual arguments. To this end, we use a general pairing function that matches (tuples of) variables with assignment expressions. Each pair forms a non-nested assignment, that further needs some provisions to handle unpacking.

Iteration The `for in` construction we have seen so far utilizes a standard *iteration protocol.* Since iteration is very common, we use different *macros* to support it, depending on the type of iterated object. For example, for non-sequence types, we use a macro that explicitly follows the iteration protocol; for sequence types, we use a macro that maintains an integer index variable.

To iterate over an integer range, the construct `for i in range(a, b)` is typically used. In this case, we use a special macro that avoids creating the range object. To iterate over a 'constant' object, such as `[(1,2),(2,3)]`, we use a sequence macro, while globally preallocating the object.

We do not follow the iteration protocol exactly, since this involves creating first-class *iterator objects*, which we currently do not support. Iterator objects are provided by iterable types, to implement the actual protocol. Therefore, we only support iteration for builtin types. Since methods such as `iter` and `xrange` return iterator objects, we have not implemented these correctly either. In case of `xrange`, we simply return a list object.

Miscellaneous Python supports *default arguments* as well as *keyword arguments*, which means actual arguments can be given in any order as long as the corresponding formals are indicated. For example, a function defined as `f(x=1,y)` can be called as `f(y=2)`. Both types of arguments are easy to support, since their usage is statically clear. However, C++ template functions cannot have default arguments, so we explicitly pass default arguments at each call site.

Finally, three major Python features can be naturally mapped to their C++ counterparts. We support *exception handling* by reusing C++ exceptions. The *namespaces* provided by Python *modules* are naturally implemented using C++ namespaces. We support *class-level attributes* by using the `static` keyword. However, we only allow the latter to be accessed, when there are no name clashes with instance-level attributes, and the class-name is explicitly given.

4.3.2 Builtin Types, Methods

To support builtin types and methods, first we must be able to *analyze* them. More specifically, for type inference to work, we need to have a description of their 'type behaviour'. We use a single Python file to *model* this behaviour. It describes the relationships between argument, return value and containee types. For example, `range` is modeled as `def range(a, b=1, s=1): return [1]`. Instance variables are used to model containee types. For example, list indexing is modeled as `def __getitem__(self, i): return self.unit`.

Secondly, we need to *implement* builtin types and methods in C++. As we never box scalar types, we can simply map these to C++ scalar types. To implement container types, we use the Standard Template Library. For example, we use `vector<T>` for sequence types, and `hash_map<K, V>` for dictionaries. However, for sake of simplicity, we keep an abstraction layer between our implementations and the STL. For example, our `list<T>` has the same set of methods as the Python list type, while using a `vector<T>` internally. Builtin methods are more straightforward to implement, and can often be bootstrapped.

To optimize our builtins, we *specialize* them for several common types, such as scalars and sequences, by using multiple signatures and template specialization. For example, consider the builtin function `list`, that converts an iterable object into a list. If the iterable object is a sequence, we can avoid following the general iteration protocol, and simply copy its contents.

String Python does not have a separate builtin type for *characters*, as these are easily represented as strings of length one. However, as many programs implicitly use characters, and a string representation is not very efficient, we need to optimize their usage. One approach is to unbox strings of length one, and use an implementation trick to identify them as such.

Instead, we use our existing type inference framework to *prove* that certain strings always have length one, potentially avoiding any overhead. We associate expressions of the form `'c'` and `s[i]`, where `s` is determined

to be of type string, with a special 'character' type, respectively before and during type inference. For each string expression, our framework automatically determines if it can only be of this type.

We map strings that are proven to be of length one to the C++ `char` type. Our existing code generation framework automatically uses this type wherever possible. For example, it maps a list of such strings to `list<char>`, which uses `vector<char>` internally.

As for integers and floats, distinguishing between characters and strings necessitates *run-time conversion* of compatible types (Section 4.2). It is even more important for characters and strings, since compatible assignments involving them are more common, and from the viewpoint of the user there really is no distinction. We use the described framework to handle conversion between characters and strings.

Tuple More so than lists, tuples are often used to hold elements of different types. Requiring these elements to have a uniform type would be a large restriction. Fortunately, tuples are usually only accessed using *constant* indices, so that we can analyze the types of individual elements. We currently only perform such an analysis for binary tuples.

During type inference, we maintain a separate instance variable for each binary tuple element, and use separate methods to access them. These methods are substituted wherever constant index values are (implicitly) used. For example, the expression `t[0]` is analyzed as `t->__getelem0__()`, where `__getelem0__` returns the respective instance variable. We also use these methods to analyze assignments such as `a,b = c`. Non-constant index expressions such as `t[x]` are analyzed as `t->__getitem__(x)`, where `__getitem__` returns both instance variables.

We do not support addition of tuples without the same uniform type, since this appears difficult to analyze and implement, and in our case would mostly result in unsupported tuples. For the same reasons, we do not support multiplication of non-uniform binary tuples.

None We map Python's None object to the *NULL-pointer* in C++, since it is typically used as such, i.e. to indicate the absence of a real object. Unfortunately, in C++ the NULL-pointer cannot be mixed with integers, as it cannot be distinguished from integer value zero. Since we never unbox scalars, and we do not wish to slow down integer arithmetic by using an extended representation, we disallow mixing of None with scalar types. One implication is that defaults for otherwise integer arguments cannot be None.

In Python, *Boolean* and *equality testing* of pointer types is more elaborate than in C++. Boolean testing of Python container types depends on whether they are empty or not. Therefore, we generally map tests on container types such as `x` to `(x && x->__truth__())`. Since equality testing can be overloaded, we generally map tests such as `a==b` to `((a&&b)?(a->__eq__(b)):(a==b))`. We simplify both types of tests to the degree that None is not involved.

Python methods return None by default. To support this, we implicitly add `return None` to methods without explicit return statements. However, since it is undecidable in general if methods reach their last statement, it is unclear what to do with methods that have return statements, but not as their final statement. We do not add an implicit return statement to such methods.

Function In C++, a *signature* has to be declared for each function reference, in the form of a *function pointer type*. Therefore, we only allow mixing of function references with equal signatures. We map each type of signature to a separate function pointer type. We have not considered using C++ generics, to support *polymorphic* function references.

Python allows for *anonymous functions*, e.g. `lambda x,y: x+y`. To support these, we create a separate function for each syntactical allocation site. We currently do not allow access to their parent scope, which is equal to their allocation scope. To support this, we would need to somehow pass necessary references at allocation time.

Class Hierarchy To support *abstract* use of builtin types, we have introduced two additional types, `pyiter` and `pyseq`, that respectively generalize builtin *iterable* and *sequence* types. On the modeling side, these are empty classes, used only to model inheritance relationships. For example, `list` inherits from `pyseq`, which in turn inherits from `pyiter`.

The added types are *data polymorphic* as well as abstract, since they generalize data polymorphic types. We do not support code generation for such types. However, this is not a problem, because we provide a manual implementation of the Python builtins. To generate type declarations for these types, we were able to use the approach described in Section 4.2, after adding some builtin-specific checks to the compiler.

On the implementation side, we use two generic base classes. The most general, `pyiter<T>`, declares virtual abstract methods, corresponding to the Python iteration protocol. The other, `pyseq<T>`, provides a `vector<T>` to contain sequence elements, and virtual methods for common sequence operations, such as `__getitem__` and `__len__`.

Actual classes inherit from these generic base classes. For example, `set<T>` inherits from `pyiter<T>`, and `list<T>` inherits from `pyseq<T>`. However, there are some complications. For example, iteration on dictionaries uses its keys, as opposed to its values. Fortunately, in C++ we can specify that `dict<K,V>` inherits from `pyiter<K>`. While `string` is further a sequence type, and it inherits from `pyseq<char>`, we use an STL `string` internally, so that we can reuse its methods. Finally, we keep binary tuples with different types of elements separate, as it is unclear how to add these to the hierarchy.

The following figure shows the complete class hierarchy for our implementation of the Python builtin types:

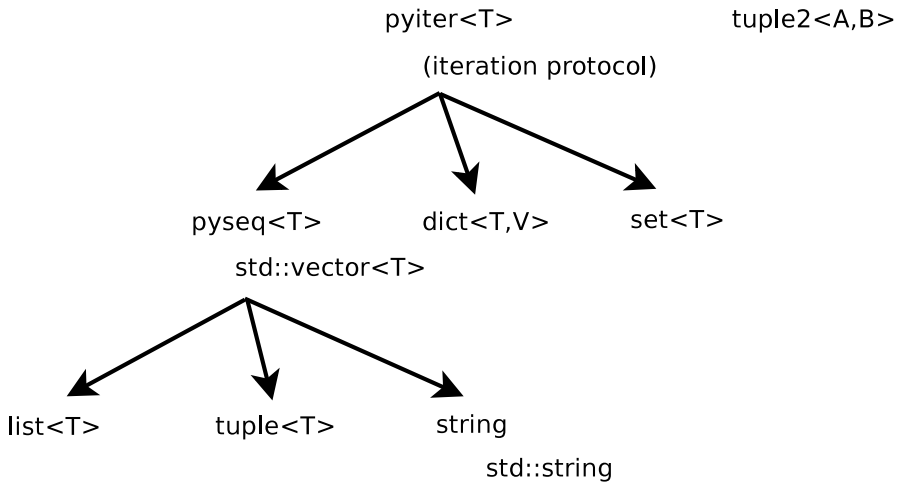

Figure 4: The complete class hierarchy for our implementation of the Python builtin types. Abstract use of binary tuples with different types of elements is currently not possible.

Builtin Methods Python comes with many useful builtin methods. For example, `zip` takes an arbitrary number of iterable arguments, and groups corresponding elements into tuples, e.g. `zip([1,2],[3,4],[5,6])` returns `[(1,3,5),(2,4,6)]`. As another example, `max` returns the maximum element of a single iterable argument, or the maximum of an arbitrary number of arguments.

We have bootstrapped the implementation of most larger builtin methods, by using our compiler on Python versions. By varying the types of arguments, we have also generated several optimized versions for sequences of scalars. Based on actual argument types, the C++ compiler automatically selects the right version.

While in general, we do not support arbitrary numbers of arguments, for builtin methods such as `zip` and `max` we currently support up to three arguments, by providing separate versions for different numbers of arguments.

4.3.3 Standard Library

Python comes with a comprehensive standard library, to perform almost any common task. For efficiency reasons, a large part of this library is written in C. Unfortunately, as it performs a *global analysis*, our compiler cannot handle such a 'black box' automatically. We circumvent the problem, by using the same technique used to support builtins.

To support (parts of) a library module, our compiler requires both a Python *model* and a C++ *implementation.* For example, `random.random` can be supported, by adding `def random(): return 1.0` to the file `random_.py`, and a C++ implementation to the files `random_.?pp`. We provide such files with our compiler, to support some of the most commonly used library methods.

5 Evaluation

In this section, we report on the *analysis time* and *run-time performance* of our compiler, for an ad hoc benchmark set of 16 non-trivial programs. We show the analysis time for different compiler settings, and compare the run-time performance with that of Psyco [17], the current state-of-the-art in optimizing Python programs, and CPython, the standard Python interpreter.

Psyco Psyco is a *specializing* JIT-compiler. It can emit multiple versions of a single piece of code, specialized to different combinations of observed variable types during run-time. It can be instructed to optimize all code, specific methods, or to use profiling to determine the most profitable methods.

A major advantage of Psyco is that it works for arbitrary Python programs. A major disadvantage is that it emits *unoptimized* machine language. Other disadvantages are that it requires CPython, typically uses lots of memory and emits x86-only machine language. It also does not *obfuscate* code the way static compilation does (this can be useful for proprietary vendors).

Setup We have used version 0.0.7 of our compiler. While this version does not optimize strings of length one (Section 4.3.2), none of our benchmarks use strings in their critical methods. To compile the resulting C++ code, we have used GCC 3.4.5, with `-O3 -s -pipe` compile flags. To perform *garbage collection* of heap allocated objects, we have used the Boehm-Demers-Weiser conservative garbage collector [3]. Finally, we have run all tests on an Athlon X2/4800+ system with 1 GB of memory, under Gentoo Linux.

For each benchmark, we show the number of lines with actual code. To give an impression of the typical compactness of Python code, we provide the full source code of the Othello benchmark in Appendix A. We distribute the source code for all benchmarks with our compiler.

5.1 Analysis Time

The following table shows the time needed to analyze each benchmark, for three different compiler settings. The first column corresponds to release 0.0.7 of our compiler. In the second and third column, we show the effect on analysis time of disabling class merging and our list heuristic, respectively (Section 4.2.1).

benchmark	loc	SS 0.0.7	-merging	-heuristic
satisfiability solver 1	81	4.92	4.12	5.91
satisfiability solver 2	122	8.92	6.26	12.64
othello player	73	3.74	3.22	11.11
neural network simulator	62	7.38	6.91	7.75
sudoku solver 1	138	8.26	6.96	14.85
sudoku solver 2	147	26.78	25.29	63.50
sudoku solver 3	154	18.73	16.57	—
n-queens problem	21	2.23	2.20	2.67
mandelbrot	25	1.89	1.81	1.82
genetic algorithm	91	3.52	3.17	3.35
pygmy raytracer	271	11.18	11.06	11.19
conway game of life	70	3.29	3.08	4.82
sea shell patterns	77	2.88	2.86	3.86
linear algebra routines	185	13.21	10.03	10.48
tic-tac-toe	112	6.30	5.17	7.41
pythonchess engine	320	14.28	12.23	15.62

Table 1: The analysis time in seconds for release 0.0.7 of our compiler, and the effect of disabling class merging and our list heuristic, respectively.

For our benchmarks, class merging invariably increases analysis time. The reason there is a negative effect at all, is that we iterate the analysis after merging. It is probably possible to avoid such iteration. Moreover, we may not see any positive effects, because the combined precision of our analysis causes Cartesian products to remain small.

On the other hand, it is clear that heuristics can have a dramatic effect on analysis time. Even with our simple list heuristic, the analysis time is more than halved, for several of our benchmarks. For the 'sudoku solver 3' benchmark, the analysis does not even terminate without it. However, this may well be caused by a bug in our compiler, or by recursive customization.

Scalability The following graph shows the relationship between analysis time and program size, based on the second column of Table 1 above (that is, for release 0.0.7 of our compiler, without class merging).

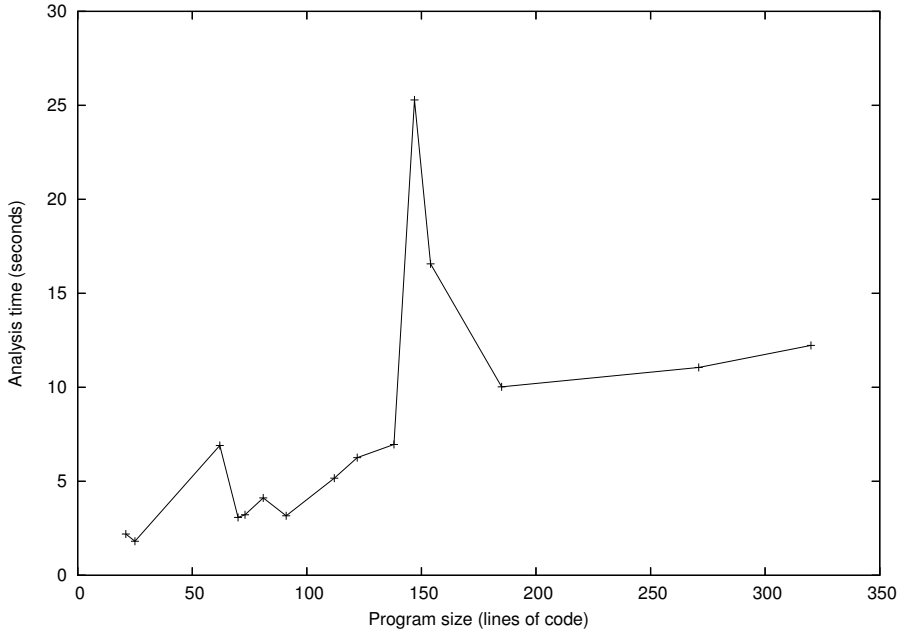

Figure 5: The relationship between analysis time and program size, for release 0.0.7 of our compiler, with class merging disabled.

For a fully adaptive approach, the analysis time should be mostly related to the 'degree' of polymorphism, rather than program size. While larger programs are typically more polymorphic, for similar amounts of polymorphism, the analysis time should not grow quickly with program size.

This is reflected in our graph. Most of the larger benchmarks are similarly polymorphic. For these, the analysis time appears to grow linearly with program size. The three benchmarks that jump out ('sudoku solver 2', 'sudoku solver 3' and 'neural network simulator') are all very polymorphic. For example, the 'sudoku solver 3' benchmark, at 154 lines of code, uses 10 different container types, 7 of which are list types. In comparison, the 'pythonchess engine' benchmark, at 320 lines of code, uses only 8 different container types, 2 of which are list types.

5.2 Run-time Performance

The following table shows the relative speedups provided by our compiler, compared to Psyco and the standard interpreter. For the first two benchmarks, we had to manually select the methods that Psyco should specialize, or it would grind to a halt.

benchmark	loc	X Psyco	X CPython
satisfiability solver 1	81	7.82	32.90
satisfiability solver 2	122	9.02	30.27
othello player	73	**34.36**	**106.21**
neural network simulator	62	7.42	13.96
sudoku solver 1	138	3.00	**58.48**
sudoku solver 2	147	2.54	4.46
sudoku solver 3	154	2.95	9.32
n-queens problem	21	1.97	36.52
mandelbrot	25	**42.90**	**228.29**
genetic algorithm	91	**12.29**	**46.14**
pygmy raytracer	271	**40.50**	**57.45**
conway game of life	70	1.30	5.88
sea shell patterns	77	**15.96**	**55.35**
linear algebra routines	185	2.56	9.37
tic-tac-toe	112	**0.75**	1.84
pythonchess engine	320	3.83	21.73

Table 2: The relative speedups provided by release 0.0.7 of our compiler, compared to Psyco and the standard interpreter.

While the results greatly vary, our compiler provides an average speedup of about 12 over Psyco and 45 over the standard interpreter. Psyco only outperforms our compiler for the 'tic-tac-toe' benchmark, and further only comes close for the 'conway game of life' benchmark.

While not shown, our simple *stack allocation* analysis significantly contributes to the speedup for six benchmarks. For these, it decreases the run time by 15 percent on average, up to 31 percent for the 'satisfiability

solver 1' benchmark. While our *static preallocation* analysis is sometimes successful, it is unable to preallocate space for any critical methods.

For most benchmarks, the speedup over Psyco can be explained by the fact that Psyco outputs unoptimized code. By adding *local optimizations*, it might compare to our compiler, for these benchmarks. However, it is unclear if it is possible or practical to add *interprocedural* optimizations to Psyco, such as stack- and static preallocation, and more traditional interprocedural optimizations as performed by our C++ compiler.

For some benchmarks, Psyco appears to get completely lost, while our compiler still provides a large speedup. For the 'othello player' and 'pygmy raytracer' benchmarks, the reason for Psyco's bad performance appears to be that these benchmarks mostly use small, *non-scalar* objects, such as binary tuples and geometric vectors, in their critical methods, and that Psyco is more effective for *scalar* computations.

On the other hand, the 'mandelbrot' and 'sea shell patterns' benchmarks are essentially simple floating-point calculations. It is unclear why Psyco does not optimize these very well. It may be that it is unable to remove all object allocations, which would greatly slow down the calculations, and that our C++ compiler is able to generate much more efficient code.

As mentioned, Psyco outperforms our compiler for the 'tic-tac-toe' benchmark. The reason for this is probably that we do not yet perform some simple source code level optimizations. For example, this benchmark uses the common `len(set(x))==1` construct, which is not very efficient if taken literally.

6 Conclusion

We have investigated the possibility of converting implicitly, but statically typed Python programs into efficient C++ code. We have described an experimental Python-to-C++ compiler, aimed at performing this task. To generate type declarations, it necessarily employs *static type inference* techniques. To avoid heap allocation if possible, it uses techniques to transform heap allocation into *stack-* and *static preallocation.*

Static type inference is difficult because of polymorphism, which causes types to be context-dependent. Two common types of polymorphism are parametric and data polymorphism. To *precisely* and *efficiently* analyze both, we have combined the Cartesian product algorithm with iterative class splitting, as suggested by Agesen [1].

We have shown that the combined type inference technique works well for a set of 16 non-trivial Python programs. It precisely determines all types, and moreover, its *analysis time* appears to depend mostly on the 'degree' of polymorphism, rather than program size. We have also shown that a simple heuristic can greatly reduce analysis time. With some further improvements, our approach can probably scale to thousands of lines of code.

We have shown that the simple invariant-based escape analysis, that we use to perform stack- and static preallocation, allows us to stack allocate a significant portion of all objects for almost half our benchmark set. We believe this indicates that a lot more is possible with more advanced memory optimizations.

Finally, we have shown that the *run-time performance* of the generated code for our benchmarks typically outperforms that of Psyco [17], the current state-of-the-art in optimizing Python programs, by a factor of 2-40. The *average* speedups over Psyco and the standard interpreter are about 12 and 45, respectively.

Given these results, we believe our original question can be answered in the affirmative:

- *It is possible to convert implicitly, but statically typed Python programs into efficient C++ code!*

At about 7,000 lines of code, our compiler has remained a relatively simple and elegant program. There are several reasons for this. First, we have combined existing, simple techniques, rather than invent our own. Second, by restricting types to be static, we have avoided several complications. Third, by actually targeting C++, we can reuse its many abstractions, as well as industrial-strength C++ compilers. Finally, of course, we have written our compiler in Python.

We have written a C++ implementation of most Python builtin types and methods, that is linked with generated code. At about 3,000 lines of code, this implementation has also remained relatively simple and elegant. One reason for this is that we can map the parametric, data and object-oriented polymorphism of these builtins to C++ *generic* and *abstract* types. The other reason is that we can delegate much functionality to the C++ Standard Template Library.

7 Future Work

In this final section, we discuss several important directions for future research. First and foremost, we need to improve the *scalability* of our combined type inference technique. Its *precision*, too, can be improved in some cases. To bring the run-time performance of generated code closer to that of manual C++ code, we have to add more advanced memory and other *optimizations*. Finally, it would be useful to make it easier for programmers to *integrate* with existing Python programs, as well as the standard library.

7.1 Scalability

The scalability of our type analysis can be dramatically improved, by providing a better initial '*guesstimate*' of actual types than our simple list heuristic. There are three ways to achieve this.

First, of course, we can improve our *heuristic*, to look more carefully at container allocation sites and their context, to see if anything hints at a certain type. We could further decrease the initial imprecision of our analysis, by partitioning the remaining allocation sites for a single class into multiple class duplicates.

Second, *profiling* can give us a very accurate view of actual types. Given a profiling technique that observes actual polymorphism, and a representative profiling run, our type analysis may only require a single iteration, as the Cartesian product algorithm keeps all actually different types apart.

Third, we can extend our analysis to work *incrementally*, by using the results of the previous analysis as the starting point for the current analysis. This could further make for a fast development cycle, without the need for any profiling, and would also be useful within an interactive development environment.

While guesstimates are very practical, they do not change the worst-case behaviour of our analysis, which should work as well as possible without them. There are several ways to improve our iterative *splitting process*.

For example, we might compare multiple confluence points, before performing splitting, and investigate different splitting heuristics. While we have not encountered it so far, we also need to investigate possible *recursive customization* (Section 3.1).

7.2 Precision

Our current type analysis is imprecise for two types of common situations. First, it does not analyze individual elements of tuples of length greater than two, or individual list elements. To support this, we can generalize our method used to analyze binary tuples (Section 4.3.2), up to some constant length.

Second, as our type analysis duplicates functions based on actual *argument types*, it fails when the types inside a function instead depend on actual *calling context* (e.g. actual call sites). For example, consider a simple function that returns an empty list. As its type does not depend on actual argument types, we cannot separate different uses of this list.

While it does not change the precision of the end result, we can also improve precision *during* the analysis. In turn, this can have a large effect on analysis time. One approach we can think of is to *filter* variable types, based on their *syntactic context*. For example, consider the expression `x.bekos()`. The variable `x` can only assume types that provide a method named `bekos`. If we enforce this, we can improve precision anywhere that `x` flows, e.g. `somefunc(x)`.

7.3 Optimization

We believe our approach can result in code that is practically as efficient as manual C++ code. However, for this to happen, we need to add and improve several analyses; we also need to improve code generation and our implementation of builtin types and methods.

Since we have focused mostly on type inference, our *stack- and static preallocation* analyses have remained very simplistic. We have mentioned several, more advanced, techniques (Section 2.2). However, there exist many more techniques to perform all sorts of *memory optimizations* [6].

There is one relatively simple technique to improve our current stack allocation analysis, and hence our static preallocation analysis. Allocated objects are typically local to the topmost few stack frames [16]. Therefore, we can improve the results of our escape analysis, by using selective *method inlining*.

In Python, using negative sequence indices has the following useful effect: `a[-x] = a[len(a)-x]`. Generated code currently has to check before each access, if the index value is negative. In many cases, it is easy to see that an index value cannot be negative. For example, a variable `i` cannot be negative, if the only assignment to it is in `for i in range(len(x))`. A simple iterative analysis can probably eliminate a large portion of all checks [6].

As for code generation, an important optimization is to always use copy-by-value for short tuples, up to a certain length, since we currently allocate all tuples on the heap. Because Python tuples are *immutable*, this cannot cause *aliasing* problems [13].

Finally, to implement strings, we currently use the inefficient STL string type. However, there are much more efficient C/C++ string libraries, such as Bstrlib [11] and the CPython implementation itself. At some point, we need to investigate the alternatives, and perhaps even create our own string library.

7.4 Integration

In its current state, our compiler is mostly useful for optimizing relatively separate, computationally intensive code. A simple way to integrate with compiled code, is to use standard in- and output, files and (in the future) sockets.

For example, a chess program might consist of two parts: a Python part, that implements the graphical interface and non time-critical functionality, and a compiled part, that evaluates the board, and communicates with the Python part via standard in- and output. This way, all code is written in Python, and the Python part can make use of the full Python semantics and the standard library.

Nevertheless, it would be useful if Python programs could be integrated more directly with compiled code, and if compiled code could make use of the full standard library. As for the first, there exist several techniques to call C++ code from a Python program, that can probably be modified to convert objects to and from our implementation of builtin types.

Integration with the standard library forms a greater challenge. Currently, a Python *model* and C++ *implementation* are necessary for each call (Section 4.3.3). Theoretically, implementations can be automatically generated, based on the respective models, and by delegating calls to the actual standard library.

References

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

[2] Agesen O. *The Cartesian Product Algorithm: Simple and Precise Type Inference of Parametric Polymorphism*, In ECOOP, p. 2-26, August 1995.

[3] Boehm H.J. *Space Efficient Conservative Garbage Collection*, In PLDI, ACM SIGPLAN Notices 28, 6, pages 197-206, 1991.

[4] Choi J.D., Gupta M., Serrano M., Sreedhar V.C., Midkiff S. *Escape Analysis for Java*, In OOPSLA, p. 1-19, November 1999.

[5] Dolby J. *An Automatic Object Inlining Optimization and its Evaluation*, In OOPSLA, p. 1-20, October 1998.

[6] Dufour M., Efficient Implementation of Modern Imperative Languages; Application to Python, `http://shed-skin.blogspot.com`

[7] Franz M., Haldar V., Krintz C., Stork C. *Online Verification of Offline Escape Analysis*, Technical Report No. 02-21, University of California, Irvine, September 2002.

[8] Gheorgioiu O., Salcianu A., Rinard M. *Interprocedural Compatibility Analysis for Static Object Preallocation*, In POPL, p. 273-284, January 2003.

[9] Grove D., Dean J., Garrett C., Chambers C. *Profile-Guided Receiver Class Prediction*, In OOPSLA, p. 108-123, October 1995.

[10] Hölzle U., Ungar D. *Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback*, In PLDI, p. 326-336, June 1994.

[11] Hsieh P., The Better String Library, http://bstring.sourceforge.net

[12] Leroy X., *The Effectiveness of Type-based Unboxing*, In Workshop on Types in Compilation, ACM SIGPLAN, June 1997.

[13] Muchnick S.S., *Advanced Compiler Design and Implementation*, Academic Press, 1997.

[14] Palsberg J., Schwartzbach M.I. *Object-Oriented Type Inference*, In OOPSLA, p. 146-161, October 1991.

[15] Plevyak J., Chien A.A. *Precise Concrete Type Inference for Object-Oriented Languages*, In OOPSLA, p. 324-340, October 1994.

[16] References Reid A., McCorquodale J., Baker J. The Need for Predictable Garbage Collection, In WCSSS, p. 56-63, May 1999.

[17] Rigo A., *Representation-based Just-in-time Specialization and the Psyco Prototype for Python*, `http://psyco.sourceforge.net/`

[18] Rossum G. van, The Python Programming Language, Community Website, `http://python.org`

[19] Veldema R.S., Jacobs C.J.H., Hofman R.F.H., Bal H.E. *Object Combining: A new Aggressive Optimization for Object Intensive Programs*, In JGI, p. 165-174, November 2002.

[20] Wang T., Smith S.F. *Precise Constraint-Based Type Inference for Java*, In ECOOP, p. 99-117, June 2001.

A Othello Player (Python)

```
empty, black, white = 0, 1, -1
player = {white:  'cpu0', black:  'cpu1'}
search_depth = 3

board = [[empty for x in range(8)] for y in range(8)]
board[3][3] = board[4][4] = white
board[3][4] = board[4][3] = black

corners = [(0,0), (0,7), (7,0), (7,7)]
directions = [(-1,-1), (-1,0), (-1,1), (0, -1), (0,1), (1,-1), (1,0), (1,1)]

# board functions

def possible_move(board, x, y, color):
    if board[x][y] != empty:
        return False
    for direction in directions:
        if flip_in_direction(board, x, y, direction, color):
            return True
    return False

def possible_moves(board, color):
    return [(x,y) for x in range(8) for y in range(8) if possible_move(board,
x, y, color)]

def flip_in_direction(board, x, y, direction, color):
    other_color = False
    while True:
        x, y = x+direction[0], y+direction[1]
        if x not in range(8) or y not in range(8):
            return False
        square = board[x][y]
        if square == empty:  return False
        if square != color:  other_color = True
        else:  return other_color

def flip_stones(board, move, color):
    for direction in directions:
        if flip_in_direction(board, move[0], move[1], direction, color):
            x, y = move[0]+direction[0], move[1]+direction[1]
            while board[x][y] != color:
                board[x][y] = color
                x, y = x+direction[0], y+direction[1]

    board[move[0]][move[1]] = color
```

```
def stone_count(board, color):
    return sum([len([square for square in line if square == color]) for line
in board])

# min-max algorithm

def best_move(board, color, first, step=1):
    max_move, max_mobility, max_score = None, 0, 0

    for move in possible_moves(board, color):
        if move in corners:
            mobility, score = 64, 64
            if color != first:
                mobility = 64-mobility
        else:
            testboard = [[square for square in line] for line in board]
            flip_stones(testboard, move, color)
            if step < search_depth:
                next_move, mobility = best_move(testboard, -color, first,
step+1)
            else:
                mobility = len(possible_moves(testboard, first))
            score = mobility
            if color != first:
                score = 64-score

        if score >= max_score:
            max_move, max_mobility, max_score = move, mobility, score
    return max_move, max_mobility

# computer versus itself

turn = black
while possible_moves(board, black) or possible_moves(board, white):
    if possible_moves(board, turn):
        move, mobility = best_move(board, turn, turn)
        flip_stones(board, move, turn)
    turn = -turn

white_stones, black_stones = stone_count(board, white), stone_count(board,
black)

if black_stones == white_stones:
    print 'draw!'
else:
    if black_stones > white_stones:  print player[black], 'wins!'
    else:  print player[white], 'wins!'
```

B Othello Player (C++)

```
#include <stdio.h>
#include "oth.hpp"

namespace __oth__ {

str *const_2, *const_3;
list<tuple<int> *> *const_0, *const_1;
dict<int, str *> *__11, *player;
int __0, __10, __7, black, depth, empty, mobility, turn, white;
tuple2<tuple<int> *, int> *__42;
tuple<int> *move;
list<list<int> *> *board;
list<int> *__5, *__6, *__8, *__9;

static inline list<int> *list_comp_0() {
    int x, __3, __4;
    list<int> *result = new list<int>();
    FAST_FOR(x,0,8,1,3,4)
        result->append(empty);
    END_FOR
    return result;
}

static inline list<list<int> *> *list_comp_1() {
    int y, __1, __2;
    list<list<int> *> *result = new list<list<int> *>();
    FAST_FOR(y,0,8,1,1,2)
        result->append(list_comp_0());
    END_FOR
    return result;
}

static inline list<tuple<int> *> *list_comp_2(int color, list<list<int> *>
*board) {
    int __24, __25, __26, __27, y, x;
    list<tuple<int> *> *result = new list<tuple<int> *>();
    FAST_FOR(x,0,8,1,24,25)
        FAST_FOR(y,0,8,1,26,27)
           if (possible_move(board, x, y, color)) {
                 result->append((new tuple<int>(2, x, y)));
             }
        END_FOR
    END_FOR
    return result;
}

static inline list<int> *list_comp_3(int color, list<int> *line) {
```

```
    list<int> *__30;
    int square;
    list<int> *result = new list<int>();
    FOR_IN_SEQ(square,line,30,31)
        if ((square==color)) {
            result->append(square);
        }
    END_FOR
    return result;
}

static inline list<int> *list_comp_4(int color, list<list<int> *> *board)
{
    list<list<int> *> *__28;
    list<int> *line;
    list<int> *result = new list<int>();
    result->resize(len(board));
    FOR_IN_SEQ(line,board,28,29)
        result->units[__29] = len(list_comp_3(color, line));
    END_FOR
    return result;
}

static inline list<int> *list_comp_5(list<int> *line) {
    list<int> *__36;
    int square;
    list<int> *result = new list<int>();
    result->resize(len(line));
    FOR_IN_SEQ(square,line,36,37)
        result->units[__37] = square;
    END_FOR
    return result;
}

static inline list<list<int> *> *list_comp_6(list<list<int> *> *board) {
    list<list<int> *> *__34;
    list<int> *line;
    list<list<int> *> *result = new list<list<int> *>();
    result->resize(len(board));
    FOR_IN_SEQ(line,board,34,35)
        result->units[__35] = list_comp_5(line);
    END_FOR
    return result;
}

int __main() {
    const_0 = (new list<tuple<int> *>(4, (new tuple<int>(2, 0, 0)), (new tuple<int>(2,
0, 7)), (new tuple<int>(2, 7, 0)), (new tuple<int>(2, 7, 7))));
```

```
    const_1 = (new list<tuple<int> *>(8, (new tuple<int>(2, 1, 1)), (new tuple<int>(2,
-1, 1)), (new tuple<int>(2, 0, 1)), (new tuple<int>(2, 1, -1)), (new tuple<int>(2,
-1, -1)), (new tuple<int>(2, 0, -1)), (new tuple<int>(2, 1, 0)), (new tuple<int>(2,
-1, 0))));
    const_2 = new str("human");
    const_3 = new str("python");

    __0 = -1;
    empty = 0;
    black = 1;
    white = __0;
    board = list_comp_1();
    __7 = white;
    __5 = board->__getfast__(3);
    ELEM(__5,3) = __7;
    __6 = board->__getfast__(4);
    ELEM(__6,4) = __7;
    __10 = black;
    __8 = board->__getfast__(3);
    ELEM(__8,4) = __10;
    __9 = board->__getfast__(4);
    ELEM(__9,3) = __10;
    __11 = (new dict<int, str *>(2, new tuple2<int, str *>(2,white,const_2),
new tuple2<int, str *>(2,black,const_3)));
    player = __11;
    depth = 3;
    turn = black;

    while((__bool(possible_moves(board, black)) || __bool(possible_moves(board,
white)))) {
        if (__bool(possible_moves(board, turn))) {
            TUPLE_ASSIGN2(move,mobility,best_move(board, turn, turn, 1),42);
            if ((!possible_move(board, move->__getfirst__(), move->__getsecond__(),
turn))) {
                printf("%s\n", "impossible!");
                turn = -turn;
            }
            else {
                flip_stones(board, move, turn);
            }
        }
        turn = -turn;
    }
    if ((stone_count(board, black)==stone_count(board, white))) {
        printf("%s\n", "draw!");
    }
    else {
        if ((stone_count(board, black)>stone_count(board, white))) {
```

```
            printf("%s %s\n", CS(player->__getitem__(black)), "wins!");
        }
        else {
            printf("%s %s\n", CS(player->__getitem__(white)), "wins!");
        }
    }
}

int possible_move(list<list<int> *> *board, int x, int y, int color) {
    tuple<int> *direction;
    list<tuple<int> *> *__12;
    if (((board->__getfast__(x))->__getfast__(y)!=empty)) {
        return 0;
    }
    FOR_IN_SEQ(direction, const_1, 12, 13)
        if (flip_in_direction(board, x, y, direction, color)) {
            return 1;
        }
    END_FOR
    return 0;
}

int flip_in_direction(list<list<int> *> *board, int x, int y, tuple<int> *direction,
int color) {
    int __14, __15, other_color, square;
    other_color = 0;
    while(1) {
        __14 = (x+direction->__getfirst__());
        __15 = (y+direction->__getsecond__());
        x = __14;
        y = __15;
        if ((!(x>=0&&x<8) || !(y>=0&&y<8))) {
            return 0;
        }
        square = (board->__getfast__(x))->__getfast__(y);
        if ((square==empty)) {
            return 0;
        }
        if ((square!=color)) {
            other_color = 1;
        }
        else {
            return other_color;
        }
    }
    return 0;
}
```

```
int flip_stones(list<list<int> *> *board, tuple<int> *move, int color) {
    tuple<int> *direction;
    list<int> *__20, *__23;
    list<tuple<int> *> *__16;
    int __18, __19, __21, __22, x, y;
    FOR_IN_SEQ(direction, const_1, 16, 17)
        if (flip_in_direction(board, move->__getfirst__(), move->__getsecond__(),
direction, color)) {
            __18 = (move->__getfirst__()+direction->__getfirst__());
            __19 = (move->__getsecond__()+direction->__getsecond__());
            x = __18;
            y = __19;
            while(((board->__getfast__(x))->__getfast__(y)!=color)) {
                __20 = board->__getfast__(x);
                ELEM(__20,y) = color;
                __21 = (x+direction->__getfirst__());
                __22 = (y+direction->__getsecond__());
                x = __21;
                y = __22;
            }
        }
    END_FOR
    __23 = board->__getfast__(move->__getfirst__());
    ELEM(__23,move->__getsecond__()) = color;
    return 0;
}

list<tuple<int> *> *possible_moves(list<list<int> *> *board, int color) {
    return list_comp_2(color, board);
}

int stone_count(list<list<int> *> *board, int color) {
    return sum(list_comp_4(color, board));
}

tuple2<tuple<int> *, int> *best_move(list<list<int> *> *board, int color,
int first, int step) {
    tuple<int> *__39, *max_move, *move, *next_move;
    list<list<int> *> *testboard;
    list<tuple<int> *> *__32;
    int __40, __41, max_mobility, max_score, mobility, score;
    tuple2<tuple<int> *, int> *__38;
    max_move = 0;
    max_mobility = 0;
    max_score = 0;
    FOR_IN_SEQ(move,possible_moves(board, color),32,33)
        if ((const_0)->__contains__(move)) {
            mobility = 64;
```

```
            score = 64;
            if ((color!=first)) {
                mobility = (64-mobility);
            }
        }
        else {
            testboard = list_comp_6(board);
            flip_stones(testboard, move, color);
            if ((step<depth)) {
                TUPLE_ASSIGN2(next_move,mobility,best_move(testboard, -color,
first, (step+1)),38);
            }
            else {
                mobility = len(possible_moves(testboard, first));
            }
            score = mobility;
            if ((color!=first)) {
                score = (64-score);
            }
        }
        if ((score>=max_score)) {
            __39 = move;
            __40 = mobility;
            __41 = score;
            max_move = __39;
            max_mobility = __40;
            max_score = __41;
        }
    END_FOR
    return (new tuple2<tuple<int> *, int>(2, max_move, max_mobility));
}

} // module namespace

int main(int argc, char **argv) {
    __shedskin__::__init();
    __oth__::__main();
}
```