Alias Analysis for Optimization of Dynamic Languages *

Michael Gorbovitski Yanhong A. Liu Scott D. Stoller Tom Rothamel K. Tuncay Tekle

Computer Science Dept., State Univ. of New York at Stony Brook, Stony Brook, NY 11794

{mickg,liu,stoller,rothamel,tuncay}@cs.sunysb.edu

Abstract

Dynamic languages such as Python allow programs to be written more easily using high-level constructs such as comprehensions for queries and using generic code. Efficient execution of programs then requires powerful optimizations—incrementalization of expensive queries and specialization of generic code. Effective incrementalization and specialization of dynamic languages require precise and scalable alias analysis.

This paper describes the development and experimental evaluation of a may-alias analysis for a full dynamic object-oriented language, for program optimization by incrementalization and specialization. The analysis is flow-sensitive; we show that this is necessary for effective optimization of dynamic languages. It uses precise type analysis and a powerful form of context sensitivity, called trace sensitivity, to further improve analysis precision. It uses a compressed representation to significantly reduce the memory used by flow-sensitive analyses. We evaluate the effectiveness of this analysis and 17 variants of it for incrementalization and specialization of Python programs, and we evaluate the precision, memory usage, and running time of these analyses on programs of diverse sizes. The results show that our analysis has acceptable precision and efficiency and represents the best trade-off between them compared to the variants.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features; D.3.4 [*Programming Languages*]: Processors—Optimization; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis

General Terms Algorithms, Languages, Performance, Experimentation

* This work was supported in part by ONR under grants N000140910651 and N000140710928 and NSF under grants CCF-0613913, CNS-0509230, CNS-0627447, and CNS-0831298.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

DLS 2010, October 18, 2010, Reno/Tahoe, Nevada, USA.

Copyright © 2010 ACM 978-1-4503-0405-4/10/10... $5.00

1. Introduction

Dynamic languages such as Python and JavaScript allow programs to be written more easily using high-level constructs such as comprehensions for queries and using generic code. Efficient execution of programs then requires powerful optimizations—incrementalization of expensive queries under updates to query parameters (Liu et al. 2005) and specialization of generic code in procedures and methods under specific calls to the procedures and methods (Rigo 2004). These optimizations require identifying values of variables and fields that are references to the same object, either a data object or a function object. Due to extensive use of object references, effective optimizations require precise and scalable alias analysis.

Alias analysis aims to compute pairs of variables and fields that are aliases of each other, i.e., that refer to the same object. Determining exact alias pairs is uncomputable (Ramalingam 1994). We use alias analysis to refer to may-alias analysis, which computes pairs that *may* be aliases, an over-approximation of exact alias pairs. An alias analysis is interprocedural if it propagates information between procedures, and intraprocedural otherwise; flow-sensitive if it computes alias pairs for each program node, and flow-insensitive otherwise; context-sensitive if it computes alias pairs for each calling context, and context-insensitive otherwise; type-sensitive if alias pairs only include variables that have compatible data types, and type-insensitive otherwise.

Making alias analysis precise and scalable is already difficult for statically typed languages, and even more difficult for dynamic languages. This is due to extensive use of features such as first-class functions, dynamic creation and rebinding of fields, methods, and even classes, and reassignment of variables to objects of different types. These features make even the construction of control flow graphs difficult. At the same time, powerful optimizations like incrementalization and specialization for dynamic languages need precise alias information at every program node and its context. Can one make alias analysis sufficiently precise and scalable for such optimizations to be effective?

This paper describes the development and experimental evaluation of a may-alias analysis for a full dynamic object-oriented language, for program optimization by incremen-

talization and specialization. The analysis has the following features:

- It is flow-sensitive. This is necessary for optimization of dynamic languages, because a variable or field may have different aliases and even different types at different program nodes, and optimizations are applied to specific program nodes. The analysis is designed by extending an optimal-time intraprocedural flow-sensitive analysis for C (Goyal 2005) to handle dynamic and object-oriented features.
- It uses precise type analysis to increase the precision of the analysis results. Precise type analysis infers not only basic types as in typed languages, but also types expressing known primitive values and ranges, and collections of known contents and lengths. These precise types are critical for handling dynamic features for constructing and refining control flow graphs in the first place. Our type analysis uses an iterative algorithm based on abstract interpretation.
- It uses a powerful form of context sensitivity, called trace sensitivity, to further improve analysis precision. It inlines all calls repeatedly except only once for recursive calls, but then merges analysis results back into the original program flow graph. This improves over flow-sensitive analysis results without needing large storage for keeping all clones, as in standard context-sensitive analysis, that help little for the optimizations.
- It uses a compressed representation for the aliases analyzed, to significantly reduce the memory used by flow-sensitive analysis. The idea is to represent aliases at a program node as differences from aliases at its control flow predecessor node if there is only one such predecessor. This is natural and simple for flow-sensitive analysis, and drastically reduces space usage.

We implemented this analysis, plus five main variations of it, for Python. The variations are:

- two flow-insensitive analyses: one that is context-insensitive, and one that is context-sensitive;
- two flow-sensitive analyses: one that is context-insensitive, and one that is context-sensitive; and
- a flow-sensitive, trace-sensitive analysis that also creates extra clones.

Each of these six alias analyses is also coupled with no type analysis, basic type analysis, and precise type analysis, resulting in a total of 18 variants of alias analysis.

We evaluate the effectiveness of these variants for incrementalization and specialization of Python programs, through applications that use InvTS, an invariant-driven program transformation system for incrementalization (Liu et al. 2005, 2009), and applications that use Psyco, a just-in-time compiler that does specialization (Rigo 2004). We also evaluate the precision, memory usage, and running time of these analyses on programs of diverse sizes. In addition, we evaluate the effect of refining control flow graphs using precise type analysis, and we examine uses of program constructs that are most challenging for precise type and alias analyses. The results show that our analysis, which is flow-sensitive and trace-sensitive and uses precise type analysis, has acceptable precision, memory usage, and running time, and represents the best trade-off between precision and efficiency for effective optimizations. For example, the analysis takes 20 minutes on BitTorrent with over 20K LOC and less than an hour on Python standard library with over 50K LOC.

A significant amount of work has been done on alias analysis, as discussed in Section 4. Our work is the first implementation and experimental evaluation of an optimal-time flow-sensitive analysis algorithm, extended to handle a full dynamic object-oriented language with precise type analysis and further improved with trace sensitivity. In contrast, almost all prior works are for statically typed languages such as C and Java. There are many uses of alias analysis for other analyses and verification, and for optimizations including specialization. Our work is the first use of alias analysis for effective incrementalization, and the first thorough evaluation of alias analysis variants for incrementalization and specialization.

Need for flow-sensitivity and type-sensitivity. We show that flow-sensitivity and type-sensitivity are essential for optimization of dynamic languages. Consider optimization by incrementalization, which replaces expensive queries with inexpensive retrievals of results that are efficiently incrementally maintained at updates to values on which the query depends. Consider the following simple example that contains updates to collections and is typical in dynamic languages as well as static languages such as Java:

```
#removes all instances of O from collection C
def removeObject(C,O):
  if isinstance(C,set):
    #a set contains O at most once,
    #thus remove it once
    if O in C:
      C.remove(O)
  if isinstance(C,list):
    #a list may contain O multiple times.
    #count the number of O's in C
    #and remove O that many times from C
    for n in range(C.count(O)):
      C.remove(O)
```

Incrementalization of a query over a collection, say `S`, typically requires insertion of maintenance code, to update the result of the query, before the removal of an element from `S`. At any statement that removes an element from any collection `C` that may alias `S`, InvTS inserts the corresponding maintenance code guarded by a runtime check that `C` is aliased to `S`. InvTS uses alias analysis to statically remove the check if `C` may be aliased to only `S`.

Suppose the alias set of `C` is `{L,S}`, where `L` is a list and `S` is a set, at the start of the body of `removeObject`. Then,

our analysis yields two different alias sets of `C`—`{S}` and `{L}`—at the two `remove` statements. This is because flow-sensitivity allows different alias sets at different nodes in the same function, and type-sensitivity uses conditions from `isinstance`. At the first `remove` statement, because `C` is aliased to only `S`, the runtime aliasing check is removed. At the second `remove` statement, because `C` may not be aliased to `S`, the maintenance code and runtime check are never inserted.

Note that for a flow-insensitive analysis of the above code, in both the original and SSA forms, the alias set of `C` is `{L,S}`. This leaves both the maintenance code and runtime check at both `remove` statements. Note also that a flow-sensitive but type-insensitive analysis would yield the same undesirable result.

2. Analysis

Our analysis takes an input program and produces information about alias pairs, as well as data types and control flows. It first handles dynamic features by analyzing types and control flows using an abstract interpretation, and then performs a flow-sensitive trace-sensitive alias analysis, or a variation of it.

The first step has two main tasks: (1) parse a program file and construct an abstract syntax tree (AST), which is easy, and (2) analyze types and construct a control flow graph (CFG) on the ASTs from all files read so far; since the code in a file may import modules from other files, analyzing a file recursively performs (1) followed by (2) at the `import` nodes. The output of this step is an interprocedural CFG of the entire program, annotated with type information.

The second step has two main tasks: (1) construct a sparse evaluation graph (SEG) from the CFG by removing CFG nodes that do not affect aliases or control flows and connecting edges to pass the removed nodes, and (2) do an alias analysis that extends an optimal-time flow-sensitive intraprocedural alias analysis to handle procedures, methods, and fields and to be trace-sensitive. We also describe a compressed representation, implementation issues, and analysis variants.

In this paper, *program node* refers to AST node. As common in languages like C and Python, *function* refers to both functions and procedures; functions are just procedures that can return values. For complexity analysis, N denotes the size of the input program, V denotes the number of variables in the program, and S denotes the maximum number of variables in scope at any program node.

2.1 Type and control flow analysis

The key challenge posed by dynamic language features is construction of a sufficiently precise CFG. Dynamic language features are: first-class functions and methods, including lambdas, inner functions, and methods in inner classes; dynamic creation and rebinding of fields, methods, and classes; reassignment of variables to objects of different types, where objects may be anything, including methods and classes; type-based dispatch, including polymorphic functions and explicit type comparison, e.g., for elements of heterogeneous collections; exceptions; and `eval`, which evaluates a string as code. These features all make it difficult to statically determine control flows.

To address this challenge, we use a precise type analysis to infer the types of variables and expressions at all program nodes, and use types to statically determine control flows as precisely as possible. In particular, dynamic features make it especially difficult to determine interprocedural control flows. Thus, we use the types of arguments and returns to help determine interprocedural control flows. We say that two types are *compatible* if their sets of possible values intersect. We add interprocedural CFG call and return edges between a call and a procedure or method only if the type signature of the call is compatible with that of the procedure or method.

Our type analysis and CFG construction is done by an abstract interpretation over a domain of precise types. It infers the types of all variables in scope at each program node, and the type of the expression at each expression node. It also constructs CFG nodes and edges as it visits program nodes following the control flows determined, easily for most intraprocedural flows, and using types for interprocedural flows and exceptions. Similarly, we use types to determine control flows involving exceptions.

Basic types and precise types. Our domain of precise types extends our domain of basic types. A precise type is a subtype of a basic type. Precise types are used in type inference and CFG construction. Basic types are used afterwards for generating specialized procedures and methods. Basic types in our type system are:

- $none$, for the special undefined value, needed in dynamic languages;
- primitive types int, $float$, and $bool$;
- collection types $string$, $list$, $tuple$, set, and $dict$ (map);
- $module$ (similar to package in Java), with, if known, module name, a list of names and their types exported by the module, and the AST node id of the module definition;
- $class$, with, if known, class name, a list of parent classes, a list of static field (including method) names and their types, and the AST node id of the class definition;
- $instance$, with, if known, type of the class of the instance, and a list of instance field names and their types;
- $function$, with, if known, function name or special name lambda (for unnamed functions), a list of parameters and their types, a list of free variables and their types (for closures), the return type, and the AST node id of the function definition;
- $method$, with, if known, everything as in function type plus the type of the instance on which the method is invoked;

- $union$, with a list of any types other than union types; union types are needed for dynamic languages, since an expression can evaluate to values of different types at different times it is evaluated; and
- bot and top, the type of no values and the type of all values, respectively; bot is a subtype of all types, and all types are subtypes of top.

Precise types extend basic types to include additional subtypes. There are three kinds of extensions:

- for primitive types, add subtypes for known values or ranges: for int, add $int_{val}(n)$ for integer constant n, int_{non_neg} for nonnegative integers, and $int_{ran}(n1, n2)$ for integers from $n1$ to $n2$, where the first of these types is also a subtype of the latter two when n is not negative or is in the range of $n1$ to $n2$, respectively; for $float$, add similar types; for $bool$, add $bool_{val}(true)$ and $bool_{val}(false)$.
- for collection types, add subtypes for known element types or lengths: for $list$, add $list_{all}[t1, ..., tn]$ for lists of known length n and element types $t1$ through tn that are not all top, $list_{len}(n)$ for lists of known length n but all top element types, and $list_{elem}(t)$ for lists of unknown length but known same non-top element type t, where the first of these types is also a subtype of the latter two when ns have the same value or $t1$ through tn are of the same type, respectively; for $tuple$ and set, add similar types; for $dict$, add similar types plus $dict_{size_key}[n, t]$ for maps of known size n, known same non-top key type t, but all top value types, and $dict_{size_val}[n, t]$ symmetrically with key and value switched, where $dict_{all}[(kt1, vt1), ..., (ktn, vtn)]$ is a subtype of both, and both are subtypes of $dict_{size}(n)$, when ns have the same value. $string$ is treated as a tuple whose element types are character types.
- for each $module$, $class$, $instance$, $function$, and $method$ type, add subtypes whose component types may use also the subtypes above, where a type $t1$ is a subtype of a type $t2$ iff all components of $t1$ are subtypes of the corresponding components of $t2$.

Any set $\{t_1, \dots, t_n\}$ of types has a minimum supertype: top if any t_i is top; otherwise union of the maximal types in all t_i if t_i's are union types, and otherwise first turn any t_i that is not a union type into a union type of itself.

We bound the set of precise types considered during type analysis to be finite, by generalizing a type to a supertype of a smaller size when the size of the type exceeds a constant. Generalization yields a minimal supertype of a smaller size; when there are multiple such types, we choose the one that merges the lowest ranges for range types, and the one with most information about element types for collection types. For example, $union(int_{val}(2), int_{val}(4), int_{val}(8))$ is generalized to $union(int_{ran}(2, 4), int_{val}(8))$ instead of $union(int_{val}(2), int_{ran}(4, 8))$, and $list_{all}[int, int, int, int, int]$ is generalized to $list_{elem}(int)$ instead of $list_{len}(5)$. The precise limit we use is for the size of each type description to be no more than 60 type names (int, $float$, etc.), except that the size of a range type is the number of times it has been generalized. Assuming that the height of the inheritance hierarchy is bounded by a constant, the number of generalizations of types for each variable is bounded by a constant. Thus, the set of types considered for each variable is bounded by a constant.

Analysis and refinement. Our algorithm does the Analysis step below to infer types and construct a CFG. Once the Analysis step reaches a fixed-point, the Refinement step below specializes the program based on the types inferred; the resulting program is then analyzed again to yield more precise types and a more refined CFG, and is analyzed incrementally. We repeat the two steps until either the resulting program cannot be further specialized, or a bound on the number of iterations is reached. The bound is set to be 30, but for all examples we have experimented with, the fixed-point was reached after 1 to 19 refinement steps, except that for Python standard library, the bound 30 had to be imposed to stop the analysis. Section 3.3 experimentally evaluates the effectiveness vs. cost of refinement.

Analysis. Start at the program entry point, and visit and interpret each program node according to its semantics in the domain of precise types. The types for all variables and expressions at all program nodes are assigned to bot initially, and go up until a fixed-point is reached. A total of 312 kinds of program nodes are handled. Most of them are for built-in functions and are obvious. We explain how the dynamic features are handled.

First-class functions and methods. At calls to first-class functions, the function type is used to determine which functions may be called. Returning, passing, or assigning a function is handled by the type analysis algorithm propagating the function type to the type of the corresponding return expression, argument expression, or the left side of the assignment, respectively. The same holds for methods.

Lambdas, inner functions, and methods in inner classes all have function types. The function type contains a list of the free variables and their types. The type is propagated by the type analysis algorithm as for other functions, and the types of the free variables are looked up when an application of the function is analyzed.

Dynamic creation and rebinding. All dynamic creation and rebinding of fields, methods, and classes are reduced to field creation and field assignment of the form `x.f=y`. Just as for normal field creation and assignment, the type analysis algorithm creates a new instance type t_{new} for `x` from the current type t_{cur} for `x`, where `f` is added to the list of fields in t_{new} if `f` is not in the list, and the type of `f` is assigned the type of `y`; the algorithm then assigns `x` the minimum supertype of t_{new} and t_{cur}.

Variables may be reassigned objects of different types, where objects may be anything, including methods and classes. This is handled by the type analysis algorithm propagating by reference, not by copying, the type of the right side of the assignment to that of the left side. Propagating the type by reference ensures that types of aliased variables change together at dynamic rebindings.

Type-based dispatch, including polymorphic functions and type comparison of elements of heterogeneous collections. At a call to a polymorphic function or method, the analysis algorithm constructs a CFG edge to each function or method with a compatible type signature for the parameters and return.

Type comparison of elements of heterogeneous collections is handled by the analysis algorithm as a normal comparison, yielding $bool_{val}(true)$ or $bool_{val}(false)$ if the types of the collection's elements are known, and are equal or not equal, respectively, and *bool* otherwise.

Exceptions. Exceptions are objects. Because `try` blocks can be nested, our analysis maintains a stack of exception handlers. When analysis enters a `try` block, it pushes on this stack the first CFG node of each `except` (similar to `catch` in Java) block together with the class types of exceptions that the `except` block handles; these stack entries are popped when analysis leaves the `try` block.

When analyzing a `try` block, including functions and methods called during it, from each CFG node n that may throw an exception, the analysis adds an edge from n to each CFG node in the stack where one of the corresponding exception class types is compatible with the type of the thrown exception, and adds an edge from n to the program exit node; to improve precision, if an exception thrown by n is definitely caught by one of the `except` clauses on the stack, edges from n to `except` clauses lower on the stack and to the program exit node are omitted. CFG edges involving `finally` blocks are added in a standard way.

Evals. The analysis distinguishes two cases. If the type of the argument of `eval` is a union of constant strings, then create a set of inner functions, one for each string in the union; create a CFG edge from the `eval` node to the entry node of each of these inner functions, and create a CFG edge from each exit node of these inner functions to the CFG node immediately following the `eval` node. The return type of the `eval` is the minimum supertype of the return types of the inner functions.

Otherwise, we use *top* as the return type. Even in this case, the behavior of `eval` of an unknown string may still be limited by the language definition; e.g., Python allows programmers to specify the sets of local and global variables that an `eval` may update. In the worst case, if an `eval` may update anything, we set the types of all variables in scope to *top* at this `eval` node; this is generally bad for precise control flow analysis, but our experiments in Section 3.4 show that this rarely occurs.

Note that imprecision caused by reflection features for accessing fields, through `setattr` and `getattr`, is limited to related objects and fields, and thus is much less problematic than `eval`.

Refinement. Refine and simplify the program using specialization and inlining as follows:

1. Clone procedures and methods so that there is one clone for each different combination of basic types of arguments a procedure or method is called with, and replace original calls with calls to the clones with the corresponding argument types.
2. Eliminate code in the clones that becomes dead for the argument types of the clone; this results in procedures and methods that are specialized for each combination of argument types.
3. Inline all procedure and method calls where inlining does not increase the number of program nodes; this eliminates the overhead of analyzing calls and returns without increasing program size.

Type and control flow analysis takes time $O(N \times S)$, because the set of types considered for each variable is bounded by a constant, and the number of refinements is bounded by a constant.

2.2 Alias analysis

Flow-sensitive alias analysis. We use the intraprocedural flow-sensitive alias analysis originally studied by Choi et al. (Choi et al. 1993), by extending the optimal-time algorithm for it by Goyal (Goyal 2005) to handle procedures, methods, and fields. The extensions are standard: treating parameter passing and result returns as assignments, and making methods into procedures that take an additional parameter for the object on which the method is invoked. We treat field dereferences as variables except that aliasing of the variable through which the field is accessed is taken into account: an assignment of the form `x.f = y` is treated as a normal update plus a weak update to `r.f` with `y` for each alias `r` of `x`; and an assignment of the form `x = y.f` is treated as a normal update plus a weak update to `x` with `r.f` for each alias `r` of `y`. The algorithm maintains a workset for each SEG node and iterates until all worksets become empty.

These extensions do not change the optimality of the time complexity. The time complexity of Goyal's algorithm is optimal because it is in the order of the size of input plus output; it is $O(N \times V^2)$ because the output is in the worst case alias pairs between all variables at each program point. The extensions do not change the order of the program size, or the number of variables; the latter is because generally there is a constant number of lexically mentioned fields relevant to each variable.

Using types to improve alias analysis precision. We modify the algorithm to only allow alias pairs that have com-

patible types. This applies to languages that do not allow arbitrary type casting, such as Python, Ruby, and JavaScript.

Our experiments show that using precise types significantly increases alias analysis precision compared with using basic types, with little or no penalty in running time.

Trace sensitivity. Precise alias analysis needs to distinguish between different calling contexts of a SEG node. We describe a new form of context sensitivity, called trace sensitivity, and compare it with traditional context-sensitive analysis.

There are two major obstacles to context-sensitive analysis. The first is recursion: the number of contexts in a recursive program may be unbounded. A standard approach to this problem is (1) representing a context as a sequence of calls or call sites, and (2) distinguishing contexts by a fixed-length subsequence of such sequences. For example, inlining n levels of function calls of the program is equivalent to (1) representing the context as a sequence of call sites, and (2) distinguishing contexts by the first n entries of the sequence — information for all contexts with the same first n call sites is merged. We refer to analysis that does 1 level of inlining as context-sensitive. Similarly, n-CFA (Shivers 1988; Vitek et al. 1992) distinguishes contexts by the last n calls — information for all contexts with the same last n calls is merged. For typical small values for n, such approaches give imprecise results for dynamic languages that routinely use double dispatch and implicit nesting of calls, such as in the case of field access in Python; larger values of n make such analyses consume an unacceptable amount of space. The second problem is that, even in non-recursive programs, the number of contexts in a program is worst-case exponential in the depth of the nested procedure calls, hence storing alias information for each context is infeasible for analyzing large programs.

We address the first problem by inlining all non-recursive calls, and by inlining calls to recursive procedures only once along a call path. We address the second problem by returning alias pairs for only nodes in the given SEG. We merge alias pairs for nodes of the inlined procedures into alias pairs for the corresponding nodes in the given SEG. We remove nodes of inlined procedures when alias pairs for them are no longer needed for the rest of the computation, reducing memory consumption.

We say that this analysis is *trace-sensitive*, because the output of the analysis depends on execution traces, but does not store information per context. Precisely, the analysis does the following:

- When encountering a call node n of a procedure f, if a clone of f is not in the current calling context of n, create a clone of f, with cloned local variables; otherwise, do analysis on the existing clone of f in the calling context.
- When adding the alias pair (x_{clone}, y_{clone}) to the alias pairs for a cloned node n_{clone}, also add the alias pair (x, y) to the alias pairs for n.
- At the end of each iteration in which an alias pair in the workset of a node n is processed, for each clone f' that is reachable from n, if the worksets of all SEG nodes that can reach the entry node of f' are empty, then f' and the alias pairs of all nodes of f' are removed to reduce memory usage.
- Perform all other operations as in the flow-sensitive algorithm described previously.
- At the end, return alias pairs for only nodes in the given SEG.

Our trace-sensitive analysis is always at least as precise as, and in our experiments always more precise than, context-insensitive analyses. The increased precision is because our algorithm distinguishes aliasing information in different contexts during analysis, even though it subsequently merges information for different contexts. Our applications in optimization do not exploit different aliasing information for different contexts.

For programs without recursion, trace-sensitive analysis is always at least as precise as, and often more precise than, an analysis that distinguishes contexts by a subsequence of the context with length n. The increased precision is because trace-sensitive analysis distinguishes aliasing information in every calling context during analysis of non-recursive programs, while an analysis that distinguishes contexts based on a subsequence of the context with length n merges aliasing information for contexts whose length is greater than n.

For programs with recursion, trace-sensitive analysis may be less precise than an analysis that distinguishes contexts based on context subsequences of length n, $n > 1$, for contexts involving recursive calls. However, in experiments we have done, an analysis that inlines n calls with $n > 1$ runs out of memory for several examples. Our experiments in Section 3.4 also show that recursion is rarely used.

We define a natural extension of trace sensitivity to allow more than one clone of a procedure in the same calling context, in essence allowing extra levels of inlining for recursive procedures. We say that an analysis is *trace-sensitive with e extra clones* if it allows $e + 1$ clones of a procedure in a calling context. We observed that for $e > 1$, the analysis runs out of memory for larger examples. We show experiments with $e = 1$ in Sections 3.1 and 3.2.

Overall, our experiments show that removing cloned procedures that can be determined to no longer alter the alias pairs is quite effective in reducing the memory usage, allowing analysis of large Python programs. Thus, trace sensitivity increases precision while remaining feasible for large programs.

Let p be the maximum size of a procedure, c be the maximum number of call nodes to a procedure, d be the maximum depth of calls to non-recursive procedures, and e be the number of extra clones allowed for each procedure. The analysis takes $O((N + (p \times c)^{d\times(e+1)}) \times (V + (p \times c)^{d\times(e+1)})^2)$ time. If one assumes that p, c, d,

and e are bounded by constants, then the time complexity of the trace-sensitive analysis is still $O(N \times V^2)$.

Compressed representation. To reduce space usage, we introduce a simple but important optimization. The alias pairs for each node that has only one control flow predecessor node are not stored explicitly, but are stored as changes to the alias pairs of the predecessor node, which themselves may be stored as changes to the alias pairs of the predecessor node of the predecessor node, all the way up to a node that has multiple predecessor nodes. A membership test against the alias pairs of a node may involve as many lookups as the length of the chain of predecessors. We bound the length of such a chain to be no more than 30. Our experiments show that this optimization reduces memory consumption for flow-sensitive analysis variants by up to a factor of 10.

Implementation issues. To implement the analyses, two additional problems must be solved.

First, non-trivial applications may use a large number of functions and classes for which the source code is not available. These functions and classes may be built into the language, be written in a different language such as assembly, or be available only in compiled form. For example, Python has over 400 special functions and classes implemented in C, either as part of the interpreter or separate C modules. The programs we analyzed contain 165 of these plus a special module. For the ten most commonly used built-in classes (`int`, `float`, `bool`, `string`, `list`, `set`, `dict`, `class`, `module`, `type`) and the special module (`__builtins__`), we laboriously hand-coded their behavior in terms of their parameter and return types, side effects, CFG effects, and effects on alias pairs, in the abstract interpreter; this took 3100 lines of Python. For all remaining 155 cases, which are the vast majority, we just duplicated the functionality of the C code in Python code without regard for time and space efficiency; this makes the implementation much easier and took only about 8000 lines of Python.

Second, the analysis on larger programs may take hours. We developed a persistence layer for the analysis framework that allows efficient storage and lookup of alias pairs on disk for further analysis. The persistence layer supports not only fast membership test against alias pairs computed by the analysis at any node, but also efficient lookup of the set of variables that a given variable aliases at a given SEG node and all of the subsequent SEG nodes in the same basic block.

To increase confidence in the correctness of the analysis and its implementation, we used `objgraph` (`http://mg.pov.lt/objgraph`) to find all references to all objects at runtime for a subset of the programs we analyzed, used this information to construct runtime alias sets for variables in the program, and then verified that the runtime alias sets are subsets of the alias sets computed by the analysis.

Analysis variants. For evaluation and comparison, we have implemented the flow-sensitive trace-sensitive analysis, plus five main variations of it, for Python. The variations are:

- two flow-insensitive analyses: one that is context-insensitive, by extending Andersen's analysis (Andersen 1994) to handle dynamic and object-oriented features in a similar way as described above, and one that is context-sensitive, by taking the flow-sensitive and context-sensitive variant below and merging the analysis results for all program nodes together.
- two flow-sensitive analyses: one that is context-insensitive, and one that is context-sensitive, both by extending Goyal's analysis as described.
- a flow-sensitive, trace-sensitive analysis that also creates extra clones.

Each of these six alias analyses is also coupled with (1) no type analysis, i.e., type-insensitive, (2) type analysis using basic types, called basic-type-sensitive, and (3) type analysis using precise types, called precise-type-sensitive, resulting in a total of 18 variants.

3. Experiments

We performed experiments that show the effectiveness of our analysis. Our first set of experiments shows that our analysis can be effectively used to incrementalize and specialize Python programs. For the trace-sensitive analysis with extra clones, we allow one extra clone. We then evaluate the precision, memory usage, and running time of analysis variants. We also evaluate the effect of refinement on alias analysis. Finally, we consider recursion, `eval`, and `exec` — constructs that can hurt our analysis precision — and show that these are rare in Python programs.

Unless otherwise specified, all experiments were performed running Python 2.6.4 on Windows 7 64bit, running on a Core 2 Quad (Q9750 at 3.8GHz) CPU with 16 GB of memory.

3.1 Effectiveness for optimization

Effectiveness for incrementalization. InvTS (Liu et al. 2005, 2009) is a transformation system for Python that performs source-level incrementalization transformations by applying transformation rules that involve alias conditions. InvTS uses alias information to statically determine the value of aliasing conditions if possible. If the value of a condition is known at compile time, InvTS can determine whether to transform a code segment. Otherwise, the condition is inserted into the generated code as a run-time check, with the transformed code in the true branch and the original code in the false branch.

InvTS experiments are conducted by transforming Python programs using transformation rules and different variants of alias analysis. The programs transformed are lxml, an XML library, and nftp, an FTP client. The transformation rules incrementally maintain properties that must hold during execution. For each analysis variant, we report the analysis time,

runtime overhead (defined as $\frac{time_t - time_o}{time_o}$, where $time_t$ and $time_o$ are the running times of the transformed and original programs respectively), and the number of alias conditions for which runtime checks are eliminated.

Lxml. Lxml (`http://codespeak.net/lxml/`) is a Python library to create and transform XML DOM trees. We applied InvTS to the test suite of the lxml library to check the following properties:

- Valid parent field: In an XML document, all non-root elements have a valid parent field, i.e., element e's `parent` field equals p iff element p has e as a child.
- No shared child and not self child: In an XML document, an element may be a child of at most one element, and an element cannot be a child of itself.
- Cause of indexing out of range: For an expression of the form `A[B]`, the value of `B` must be a valid index of `A`. If this property is violated, report the files and lines where the `index out of bounds` exception became unavoidable, i.e., the location at which each variable that was in the expression that eventually caused the exception was last modified during execution so as to cause the exception.

The test suite processed 10 million XML records.

Table 1 shows that the overhead of maintaining these properties, when the transformation uses a flow- and context-sensitive but type-insensitive analysis, is 83%, 93%, and 310%, respectively. Precise type sensitivity decreases these to 73%, 89%, and 192%. Adding trace sensitivity further decreases the overhead to 14%, 85%, and 85%, but increases the analysis time by up to 41% (from 61 to 86 seconds).

Nftp. Using InvTS, we found the cause of a previously encountered bug in nftp (`http://inamidst.com/proj/nftp`), an FTP client that downloads directories from multiple machines. This bug occurs when a directory listing command is issued before a change directory command completes. We wrote a transformation rule that maintains a set of outstanding FTP commands, and uses it to determine where this error occurs. We ran nftp with 10 threads, with 30 directories totaling 20GB over a 1GBit connection, ensuring that the program is CPU bound. Table 1 shows that the runtime overhead when using flow- and context-sensitive but type-insensitive analysis is 91%. Adding precise type sensitivity reduces it to 81%, and adding trace sensitivity further reduces it to 73%.

General observations. Table 1 summarizes our InvTS experiments. Figure 1 shows the overhead for the six precise-type-sensitive variants of alias analysis. Flow-insensitive analysis performs poorly, whether context-sensitive or not. For flow-sensitive analysis variants, the context-sensitive analysis performs only slightly better than the context-insensitive analysis. A reason for this is, in Python, field assignments are usually two nested calls (`__setattr__` and `__setitem__`; `__setattr__` is a method of the object being updated, which usually then calls the `__setitem__` method of the `dict` class object that represents fields of an object as key-value pairs).

In general, n levels of inlining with the typical small values for n give imprecise results for dynamic languages that routinely use double dispatch and implicit nesting of calls, such as in the above case of field access in Python; larger values of n make the analyses consume an unacceptable amount of space.

We conclude that the best trade-off between precision and analysis time is the flow-, trace- and precise-type-sensitive analysis. While adding extra clones slightly increases precision, it takes several times as long to run.

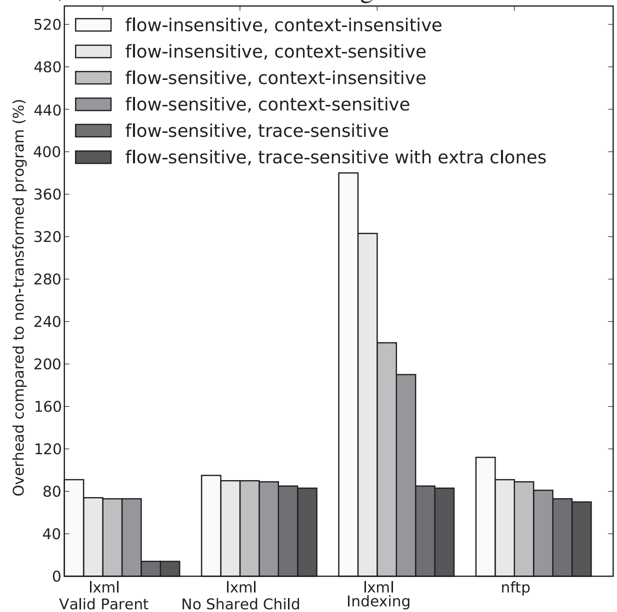

Figure 1. Runtime overhead of transformed programs, using precise-type-sensitive alias analysis, varying flow and context sensitivity.

Effectiveness for specialization. Psyco (Rigo 2004) is a specializing just-in-time compiler for Python. At startup, it compiles all the bytecode it can to machine code. The remaining bytecode needs more information to be compiled, including alias and type information. After collecting more information at runtime, Psyco compiles the remaining code to machine code. In our experiments, we augmented Psyco so it can use statically computed alias and type information. Specifically, we modify Psyco's unlift operator as follows: when Psyco blocks specialization of a function until a runtime answer to the question "is x the same as y" or "is x of type T" becomes available, and the answer is available from statically computed alias or type information, we return that answer to Psyco, and Psyco continues specializing the function. This allows Psyco to compile functions at startup that otherwise it would have to compile at runtime after collecting more information.

We ran Psyco on its largest included benchmark, which consists of 397 lines of code, and performs assignments,

flow sensitive	context sensitivity	type sensitivity	lxml - Valid Parent 97 alias checks			lxml - No Shared Child 81 alias checks			lxml - Indexing 1451 alias checks			nftp 31 alias checks		
			runtime overhead	checks removed	analysis time	runtime overhead	checks removed	analysis time	runtime overhead	checks removed	analysis time	runtime overhead	checks removed	analysis time
no	no	no	92%	12	36	95%	12	39	440%	35	49	119%	7	19
		basic	93%	12	36	95%	13	38	429%	35	50	119%	7	19
		precise	91%	14	36	95%	13	39	381%	41	49	112%	9	19
no	yes	no	88%	16	60	94%	15	62	364%	55	97	110%	9	83
		basic	88%	17	64	93%	17	62	350%	61	97	96%	11	82
		precise	74%	26	61	90%	23	61	323%	89	99	91%	13	84
yes	no	no	87%	17	42	93%	19	42	340%	79	62	93%	12	30
		basic	86%	17	43	91%	20	43	331%	81	61	89%	13	30
		precise	73%	28	43	90%	28	46	219%	122	61	89%	13	30
yes	yes	no	83%	18	59	93%	20	57	310%	103	98	91%	13	80
		basic	82%	18	61	90%	23	63	303%	112	95	86%	14	82
		precise	73%	30	61	89%	29	61	192%	199	98	81%	14	81
yes	trace	no	82%	20	81	91%	19	85	160%	246	103	90%	12	63
		basic	75%	28	82	88%	28	85	133%	344	109	77%	14	62
		precise	14%	68	82	85%	40	86	85%	836	104	73%	16	63
yes	trace extra	no	67%	37	308	85%	37	312	124%	455	783	78%	14	119
		basic	19%	61	308	85%	38	310	99%	603	780	74%	15	119
		precise	14%	72	310	83%	41	311	83%	892	791	70%	17	118

Table 1. Runtime overhead, number of alias checks removed, and analysis time (in seconds) in InvTS experiments. Runtime overhead is $\frac{time_t - time_o}{time_o}$, where $time_t$ and $time_o$ are running times of the transformed and original programs, respectively.

class construction, function and method calls, and list and dictionary operations. For this benchmark, Psyco, with no additional alias information passed to it, compiles only 43 out of 73 procedures at startup, speeding the program up 44%. We provided the results of each alias analysis variant to Psyco, and measured the number of non-compiled procedures and the speedup compared to Psyco run without this information. We do not include analysis time when computing speedup, because analysis information can be computed once per program.

Table 2 shows that the number of procedures compiled at startup, and the resulting speedup, increases with the precision of the alias analysis and type sensitivity. Flow-, trace- and precise-type-sensitive analysis with extra clones yields the best results, a speedup of nearly 16% compared to the original Psyco, which is 53% when compared to Python without Psyco, computed as $1-(1-0.44)\times(1-0.16)$. Eliminating the use of extra clones reduces the speedup by 0.4% (15.9% - 15.5%) and the analysis time by 84% ($\frac{339.3-52.6}{339.3}$). Even though the analysis time is significant, doing the analysis is worthwhile because after performing the analysis just once, every future run of the program can use the analysis results without performing the analysis again, thus amortizing the cost of one analysis over a potentially very large number of runs.

flow sensitive	context sensitivity	type sensitivity	program speedup	uncompiled procedures	analysis time
no	no	no	3.8%	27	1.8
		basic	4.8%	26	1.9
		precise	6.7%	23	2.2
no	yes	no	7.2%	24	26.6
		basic	7.7%	23	26.9
		precise	10.9%	21	27.0
yes	no	no	7.2%	25	4.0
		basic	7.2%	23	4.1
		precise	11.3%	20	4.2
yes	yes	no	6.7%	24	23.1
		basic	7.7%	23	24.1
		precise	13.4%	18	23.8
yes	trace	no	8.2%	24	51.1
		basic	10.0%	22	51.4
		precise	15.5%	16	52.6
yes	trace extra	no	9.9%	22	331.1
		basic	11.3%	20	335.7
		precise	15.9%	15	339.3

Table 2. Program speedup, number of procedures left uncompiled at compile-time, and analysis time (in seconds) in Psyco experiments. Program speedup is $\frac{time_o - time_a}{time_o}$, where $time_a$ is the running time using Psyco with alias information, and $time_o$ is the time using the original Psyco, which leaves 30 procedures uncompiled.

3.2 Precision, memory usage, and running time

We evaluated the precision, maximum memory usage, and running time of the analysis variants by running them on seven Python programs of diverse sizes. The programs include the standard Python (`http://www.python.org`) modules `chunk`, `bdb`, `pickle`, and `tarfile`; `Fortran2003`, a module of `SciPy` (`http://www.scipy.org/`); `bitTorrent` (`http://www.bittorrent.com/`); and `std. lib.`, the set of Python standard libraries used by the pro-

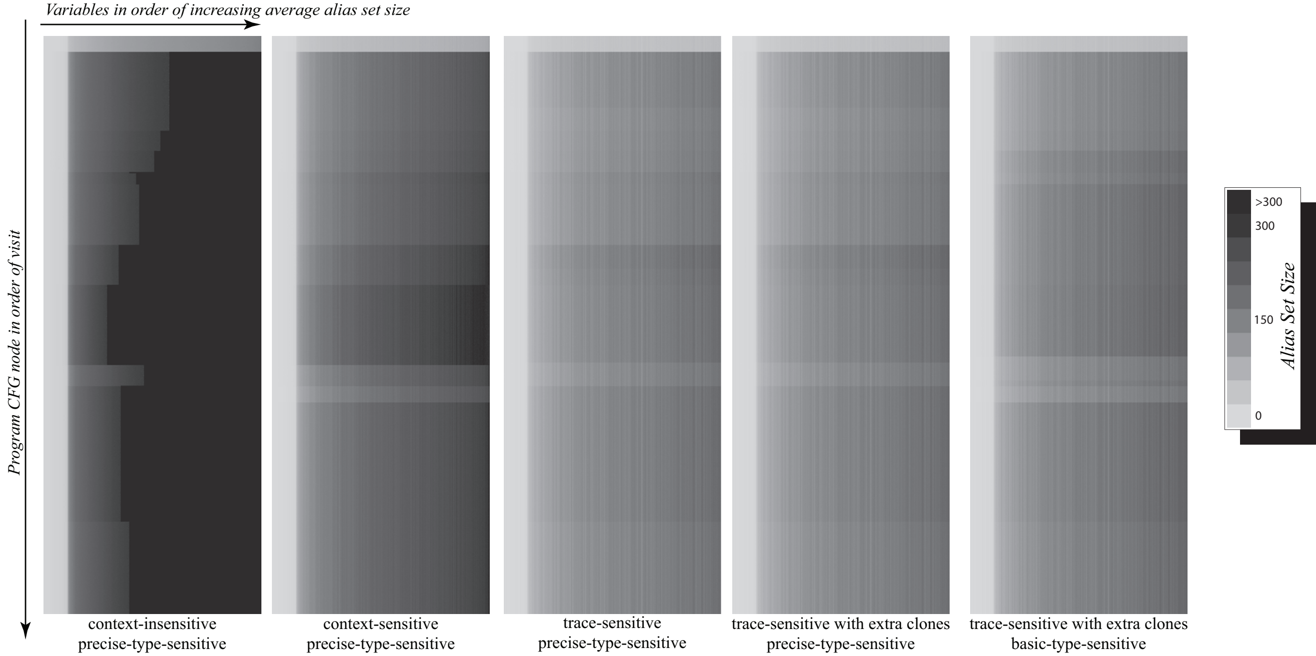

Figure 2. Alias set size for each variable (shown horizontally) for each CFG node (shown vertically) for flow-sensitive analysis variants for `tarfile`. Variables are ordered by increasing average alias set size in the context-insensitive precise-type-sensitive analysis.

grams we analyzed. We recorded the output, running time, and maximum memory consumption.

Precision of alias analysis variants. Figure 2 shows a visual comparison of the results of the alias analysis of `tarfile`, for four flow- and precise-type-sensitive analysis variants, plus, for comparison, the trace- and basic-type-sensitive analysis with extra clones. Columns represent the variables in the program; rows represent the CFG nodes. The shading represents the size of the alias set of a variable at a CFG node, where the alias set of a variable is the set of variables it may alias; lighter colors represent higher precision, and darker colors represent lower precision. This graph makes it clear that as we add context- and trace-sensitivity, the precision of the analysis increases. Adding extra clones also improves precision, but not by as great an extent. Type insensitivity reduces the precision of the analysis. Trace-sensitive analysis with extra clones takes far more time than trace-sensitive analysis without extra clones, while providing only slightly higher precision. We conclude that the most practical alias analysis is the flow-, trace-, and precise-type-sensitive analysis.

Memory usage. Figure 3 shows the memory usage of the four flow- and precise-type-sensitive analysis variants, with and without compressed representation, and of the two uncompressed trace-sensitive variants without trace optimization (removal of no longer needed procedure clones). Due to the large spread of values, both axes are drawn in log-scale.

Despite being a smaller program, the memory usage for several variants of the analysis of `tarfile` is larger than for `Fortran2003` because the average size of alias graphs in `tarfile` is significantly larger when analyzed by a flow-sensitive analysis. The memory usage for flow-insensitive analysis variants are not shown because they are much smaller.

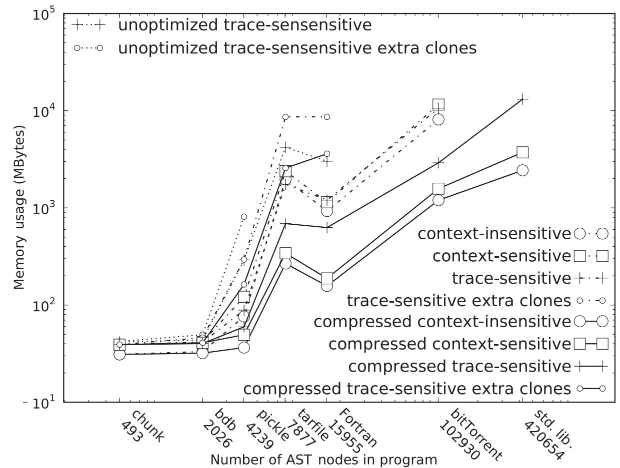

Figure 3. Maximum memory usage for flow- and precise-type-sensitive alias analysis variants, varying context sensitivity using uncompressed or compressed representations. "unoptimized" means that trace optimization and compression are both disabled; trace optimization is enabled for all other trace-sensitive variants. Data points are missing for cases where the analysis ran out of memory or time (limited to 4 hours). Both axes are log scale.

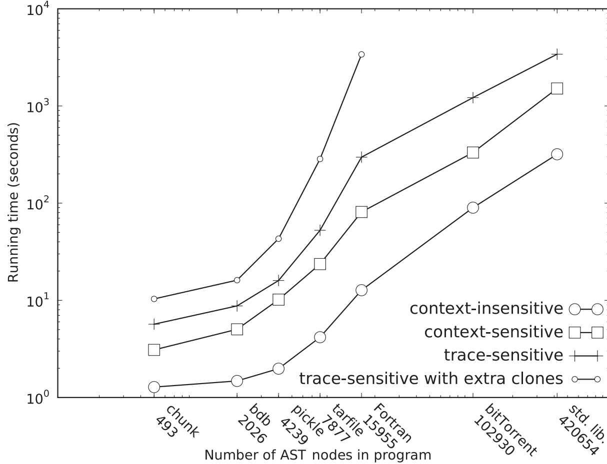

Figure 4. Running times for flow- and precise-type-sensitive alias analysis variants using compressed representation, varying context sensitivity. Both axes are log scale.

From Table 3, it is clear that for trace-sensitive analysis of large programs, both trace optimization and compressed representation are required, otherwise memory usage is prohibitively large on even medium-sized programs such as `tarfile`. Analyzing `tarfile` without the optimizations consumes over 4 GB of memory. Trace optimization alone reduces this to a still large 1.75 GB, while increasing running time by 46%, from 31.36 seconds to 45.90 seconds. Combining trace optimization and compressed representation further reduces the memory usage to 0.69 GB, while increasing the running time by only 14%, from 45.90 seconds to 52.38 seconds. Combining these two optimizations makes it feasible for trace-sensitive analysis to analyze `bitTorrent` and `std. lib`.

Running time. Figure 4 and Table 3 show the running time of the four flow- and precise-type-sensitive analysis variants, using compressed representation, and where applicable, trace optimization. For example, on BitTorrent with over 20K LOC, our flow-sensitive, precise-type-sensitive, and trace-sensitive analysis that uses compressed representation takes 20 minutes and 12 seconds.

Here again, the trace-sensitive analysis is the most precise feasible variant, as the trace-sensitive variant with extra clones takes almost 1 hour to complete on `Fortran2003`, and times out (exceeds 4 hours) on `bitTorrent` and `std. lib`. Without extra clones, the trace-sensitive analysis takes less than an hour to analyze `std. lib.` with over 50K LOC. Running times of type-insensitive and basic-type-sensitive alias analysis variants are not presented because in our experience, increasing type sensitivity does not significantly increase alias analysis time, especially when compared to the benefits of precise type sensitivity. Table 1 shows this: the largest slowdown caused by precise type sensitivity is eleven seconds (`lxml - Indexing`, trace- and precise-type-sensitive vs. trace- and basic-type-sensitive variant), and there are cases where precise type sensitivity actually speeds alias analysis up.

Table 3 shows the data used to generate Figures 3 and 4.

	without refinement	with refinement
MAASS, all variables	15.3	15.1
MAASS, locals and parameters	4.7	2.8
number of AST nodes	5021	5619

Table 4. Precision of alias analysis of `lxml - Indexing`, with and without refinement. 12 refinement steps are performed before a fixed-point is reached. MAASS is the mean average alias set size of variables in specialized functions, computed as described in text.

3.3 Effect of refinement on alias analysis

In this section, we determine the effect of refinement on alias analysis, and show that refinement is worthwhile. To do this, we perform the following experiments on a subset of programs from Section 3.1:

- We measure the effect of refinement on the precision of alias analysis results.
- We measure how the program size varies as a function of the bound on the number of iterations of analysis and refinement.
- We measure how the overhead of the programs transformed by InvTS varies as a function of the bound on the number of iterations of analysis and refinement.
- We measure how the time taken to transform these programs varies as a function of the bound on the number of iterations of analysis and refinement.

Effect of refinement on precision of alias analysis. To demonstrate how the precision of alias analysis results changes due to refinement, we performed alias analysis on `lxml - Indexing` program, without refinement, and then with refinement until a fixed point was reached. This resulted in 7 functions being specialized into 19 functions. We compute an average alias set size for each variable used in these functions, by averaging the alias set size for that variable at all of the AST nodes in the functions. We then compute the mean average alias set size (MAASS) by taking the mean of the average alias set size for a set of variables. We compute the MAASS first over all variables, then over a subset consisting of only local variables and formal parameters.

Table 4 presents the results of this experiment. Using refinement introduced 598 new AST nodes. Adding these nodes allowed the refined functions to be analyzed more precisely, with the MAASS decreasing from 15.3 to 15.1. When only local variables and parameters are considered, the MAASS was reduced more substantially, from 4.7 to 2.8. This shows that refinement is effective at decreasing the alias set size of local variables and parameters.

Effect of refinement on program size. Refinement specializes functions before alias analysis is performed, so it may

Program	LOC	AST Nodes	context-insensitive unoptimized time	memory	context-insensitive uncompressed time	memory	context-insensitive compressed time	memory	context-sensitive unoptimized time	memory	context-sensitive uncompressed time	memory	context-sensitive compressed time	memory
chunk	172	493	not applicable		1.01	31.06	1.28	31.04	not applicable		2.58	39.07	3.10	39.07
bdb	609	2026			1.20	33.25	1.48	32.03			4.52	41.71	5.07	40.85
pickle	1392	4239			1.65	76.20	1.98	36.51			10.04	121.43	10.11	49.48
tarfile	1796	7877			3.23	1964.09	4.16	267.70			20.69	2384.95	23.11	341.45
Fortran	6503	15955			11.94	928.16	12.77	157.25			77.71	1142.45	80.97	188.16
bitTorrent	22423	102930			63.01	8134.75	90.01	1198.93			298.86	11555.96	330.44	1574.81
std. lib.	51654	420654			out of memory		317.44	2434.01			out of memory		1519.68	3726.77

Program	LOC	AST Nodes	trace-sensitive unoptimized time	memory	trace-sensitive uncompressed time	memory	trace-sensitive compressed time	memory	trace-sensitive with extra clones unoptimized time	memory	trace-sensitive with extra clones uncompressed time	memory	trace-sensitive with extra clones compressed time	memory
chunk	172	493	4.09	41.74	4.97	39.16	5.65	39.13	7.10	42.26	8.89	39.26	10.37	39.15
bdb	609	2026	7.60	43.76	7.61	41.40	8.76	40.18	12.90	49.46	13.91	46.15	16.08	40.85
pickle	1392	4239	11.12	291.61	13.94	88.60	15.97	59.74	21.11	812.11	34.69	294.06	43.13	162.91
tarfile	1796	7877	31.36	4203.29	45.90	1751.84	52.38	688.53	out of memory		236.76	8631.85	283.45	2570.28
Fortran	6503	15955	123.65	3018.57	262.93	1202.04	298.23	627.41	out of memory		2687.26	8645.29	3389.17	3602.21
bitTorrent	22423	102930	out of memory		1068.36	10618.39	1211.87	2909.11	out of memory		out of time		out of time	
std. lib.	51654	420654	out of memory		out of memory		3401.69	13124.52	out of memory		out of time		out of time	

Table 3. Running time (in seconds) and maximum memory usage (in MBytes) for flow- and precise-type-sensitive alias analysis variants. "unoptimized" means that trace optimization and compression are both disabled; trace optimization is enabled for all other trace-sensitive variants; "not applicable" means that trace optimization is not applicable to trace-insensitive variants; "out of memory" means that the memory usage of the analysis exceeded 16 GB; "out of time" means that its running time exceeded 4 hours.

increase the size of the program that the alias analysis has to analyze. We quantify this increase by measuring the number of AST nodes after refinement as a function of the bound on the number of iterations of analysis and refinement. Figure 5 shows that for all programs from Section 3.1, the program size never increases more than 11%. For programs from Section 3.2, refinement increased the number of AST nodes of the analyzed program by an average of 13.6%; the maximum increase was 28.6%, for Python standard library.

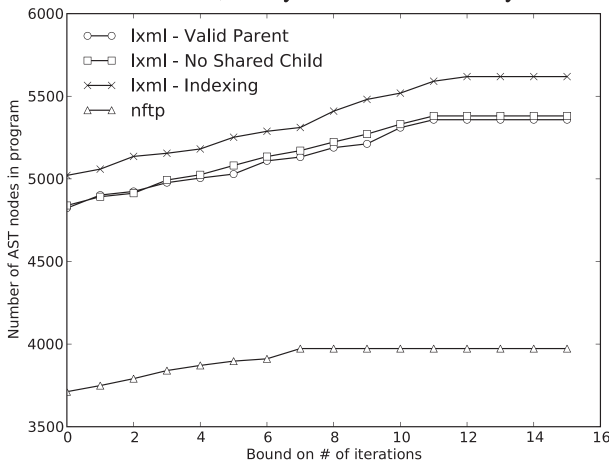

Figure 5. Number of AST nodes, as a function of the bound on the number of iterations of analysis and refinement.

Effect of refinement on optimization. The increase in program size due to refinement potentially increases the alias analysis time. To determine whether the cost of refinement is worthwhile, we measured (1) how the overhead of the programs transformed by InvTS in Section 3.1 varies as a function of the bound on the number of iterations of analysis and refinement, and (2) how the total transformation time (including analysis time) for these programs varies as a function of that bound. The experiments were performed using the same setup as the experiments in Section 3.1.

Figure 6 presents the results. For each program, overhead decreases as the bound increases, up to the point where a fixed-point is reached, i.e., further iterations of analysis and refinement do not specialize any more functions. For `lxml - Indexing` and `nftp`, this happens when the bound is higher than 12 and 7, respectively. The overhead reduction is in some cases quite significant, such as the almost 20% reduction for `lxml - Indexing`; the extra transformation time due to refinement never exceeds 10 seconds, i.e., 12% of the total transformation time. Thus, for InvTS, refinement is clearly worthwhile, especially since the relatively minor refinement cost is incurred just once, but the benefits of lower overhead are reaped every time the transformed program is executed.

3.4 Prevalence of recursion, eval, and exec

Recursion. Trace sensitivity is a good fit for programs where deeply nested function calls are common, and recursion is not prevalent. To determine how common recursion is in Python programs, we looked at all Python programs (`.py` files) on an Ubuntu 8.10 system, a total of 7,740 programs, including Python 2.4 and 2.5 standard libraries, the `zope` framework, and many other utilities and libraries.

We statically analyzed these programs to detect the presence of recursion that involves only calls to functions and calls to methods through `self`, analogous to `this` in Java. Specifically, we parsed the program and constructed a call graph whose nodes are fully qualified function or method names, and with call edges induced by function calls and method calls through `self`, i.e., calls of the form `self.m(...)` (this is a call to the method `C.m`, where `C` is the

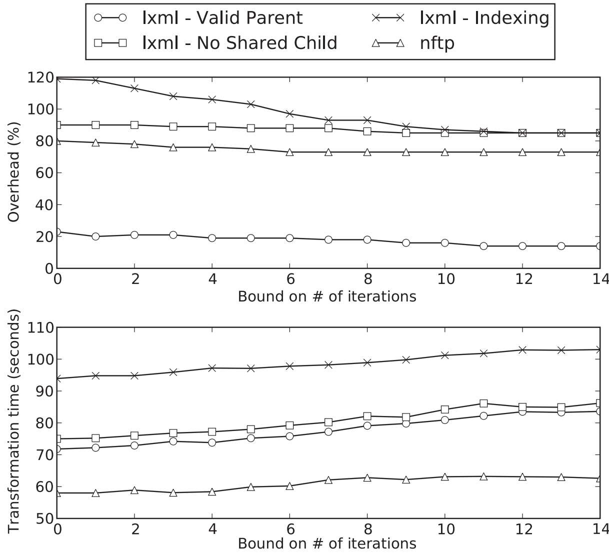

Figure 6. Runtime overhead of transformed programs and total program transformation time, using precise-type-sensitive alias analysis, as a function of the bound on the number of iterations of analysis and refinement.

enclosing class). The call graph was searched for strongly connected components (SCCs), which indicate recursion.

This analysis detected recursion in 461 out of 7,740 programs analyzed. Specifically, 738 out of a total of 264,080 functions are in strongly connected components.

Since this analysis may miss some recursions, and it may report recursions that rarely (or never) occur during execution, we also performed runtime detection of recursion on a subset of the programs. Specifically, we ran the program in a way that recorded the call history, and detected cycles in the call history; these cycles indicate recursion. Out of the 7,740 programs, we selected ones with a history of more than 50 calls when run without arguments. This eliminated programs that trivially terminate, and left 974 programs.

Analysis of the call histories detected recursion in 66 of these 974 programs. Our static analysis detected recursion in 64 of these 66 programs; this is an encouraging level of agreement. If the programs surveyed are representative, our results show that the use of recursion in Python programs is limited.

Eval and exec. Uses of `eval` functions and `exec` statements (which are similar to `eval` functions, but do not return values) cause the type of all accessible variables to become *top*. This can be detrimental to the precision of the type analysis unless the calls to `eval` or `exec` contain a scope argument that restricts the set of accessible variables.

We found that 237 out of the 7,740 programs use `eval` or `exec`, and only 39 of them do not restrict the set of accessible variables.

To determine how frequently `eval` or `exec` are called, we performed an experiment similar to the one for runtime recursion detection, except that we searched the call histories for calls to `eval` or `exec`. Out of the 974 programs analyzed, only 101 use these constructs outside of the Python libraries we reimplemented. Our reimplementations do not use `eval` or `exec`. Thus, for the purposes of our type analysis, calls to `eval` or `exec` occurred in approx. 10% of the programs surveyed.

Using our type analysis to determine all possible targets at function call sites and method call sites, we statically detected all direct and indirect uses of `eval` and `exec` in the programs from Sections 3.1 and 3.2. We manually inspected uses of these constructs to determine whether the set of accessible variables is restricted. We found that only `bdb` uses `eval` without restricting the set of accessible variables; `Fortran2003`, `InvTS`, and `std. lib.` use `eval` but restrict the set of accessible variables; `chunk`, `pickle`, `tarfile`, all `lxml` programs, `nftp`, and `bitTorrent` do not use these constructs at all. This confirms that use of `eval` or `exec` with no restriction on the set of accessible variables is rare in Python programs.

4. Related work

Alias analysis and the related problem of points-to analysis have been studied extensively (Hind 2001), mostly for statically typed languages, such as C and Java. Many positions on the spectrum of trade-offs between precision and scalability have been explored: flow-insensitive, context-insensitive analyses, such as (Andersen 1994; Steensgaard 1996); context-sensitive, flow-insensitive analyses, such as (Foster et al. 2000; Fähndrich et al. 2000; Milanova et al. 2005); context-insensitive, flow-sensitive analyses, such as (Choi et al. 1993; Goyal 2005); and context-sensitive, flow-sensitive analyses, such as (Vitek et al. 1992; Emami et al. 1994).

There have been some studies on these trade-offs in the context of statically typed languages. For example, flow sensitivity in analysis of C programs provides little improvement in precision for some applications (Hind and Pioli 2001; Mock et al. 2002) but is important in others (Hardekopf and Lin 2009); similarly, context sensitivity provided little precision benefit in analysis of some C programs (Ruf 1995) but was significant for some Java applications (Lhoták and Hendren 2006).

Our analysis is trace-sensitive, a form of context sensitivity based on cloning of functions. Guyer and Lin's client-driven pointer analysis for C also uses cloning in providing a customizable level of context sensitivity to client analyses (Guyer and Lin 2005). Significant differences between their work and ours are the target language (C vs. Python) and the client analyses considered (error detection vs. optimization). Lattner et al. use a form of context sensitivity that collapses strongly connected components and then inlines everything (Lattner et al. 2007). Their analysis is for C and is flow-insensitive, hence not appropriate for the optimizations we consider as clients.

Sridharan and Bodik's analysis (Sridharan and Bodik 2006) also collapses strongly connected components in a context-sensitive points-to analysis for Java, but the analysis mutually refines call graphs and points-to information, while also filtering out unrealizable paths based on queried variables, making the analysis more scalable than possible before. The analysis is still flow-insensitive and does not handle many dynamic features that we handle, and thus still leaves much to be desired in precision and scalability for optimization of dynamic languages.

We believe that trace-sensitive analysis is especially suited for optimizations, for both dynamic languages and static languages.

Our work is the first to assess the impact of flow sensitivity, context sensitivity, and type sensitivity on precision, memory usage, and running time of alias analysis for a dynamic object-oriented language, and evaluate the effectiveness of these analyses for program transformations and optimizations. We give a simple example that shows flow-sensitivity and type sensitivity are essential for a precise analysis and effective optimization, whereas a context-insensitive or context-sensitive analysis over an SSA representation (Hasti and Horwitz 1998; Bravenboer and Smaragdakis 2009) does not give the precision needed for optimization. There are fast and scalable context-sensitive but flow-insensitive analyses (Bravenboer and Smaragdakis 2009), but flow- and context-sensitive analysis of dynamic languages presents unique challenges, e.g., significantly larger memory footprint and many more strong updates.

Previous work on alias analysis for dynamic object-oriented languages does not handle the breadth of dynamic features that we handle. For example, the alias analysis for PHP in (Jovanovic et al. 2006; Balzarotti et al. 2008) does not handle first-class functions (which PHP does not support) or `eval` statements, and does not compare different variants of the analysis. Jang and Choe (Jang and Choe 2009) handles only a simple subset of JavaScript.

Control flow analysis for dynamic languages has been used for Ajax intrusion detection (Guha et al. 2009). In that work, an interprocedural CFG for a JavaScript program is constructed using k-CFA and then transformed into a request graph to build an intrusion-detection proxy for the server that the program communicates with. Similar to our type analysis, their analysis tracks constant strings and string operations and allows static evaluation of `evals` on constant strings. They make assumptions regarding `evals` that we do not make: `evals` return only objects that do not have methods, and `evals` do not write into variables that are not local to the argument of the `eval`. Their analysis is reported to take about 45 minutes on programs of about 6K LOC but cannot handle 10K LOC.

Type analysis for dynamic languages is well known to be difficult. Starkiller (Salib 2004), a static type inference engine for Python, has several limitations compared to our work: it is flow-insensitive (i.e., does not allow variables to have different types at different program nodes), does not support union types, and does not track contents of collections. The type system and type inference algorithm for a subset of JavaScript in (Anderson et al. 2005) also has these limitations; in addition, it does not support field and method names as strings, functions as expressions, or `eval`. Localized Type Inference (Cannon 2005) for Python cannot infer types of method and procedure arguments automatically, and does not support single-value types, range types, or union types. DiamondBack (Furr et al. 2009), a static type inference system for Ruby, supports intersection types, union types, single-value types, and parametric polymorphism, but it does not support analysis of `eval` or method calls when the target object's type is unknown. Our precise types for Python are sketched briefly in (Gorbovitski et al. 2008), but it does not describe handling of dynamic language features, generalization during type analysis, and refinement between analysis.

Our static type analysis plays two important roles. First, type information is used to statically determine dynamic dispatch, which is crucial to obtain a precise control flow graph (Bacon and Sweeney 1996; Sreedhar et al. 2000). Second, type information is used to eliminate alias pairs that are impossible due to type mismatches. Type information has been used for the latter purpose in alias analysis for statically typed languages, e.g., Modula-3 (Diwan et al. 1998) and Java (Lhoták and Hendren 2003), but it does not significantly help there, because most statements that would create such alias pairs are rejected by the type checker. In contrast, our experiments show that static inference of precise types provides significant benefits for alias analysis for dynamic languages.

Storing all of the alias sets for a program can consume a lot of memory, especially for flow-sensitive, context-sensitive analyses. We reduce the memory requirements using a compressed representation that exploits the similarity between alias sets at adjacent nodes in the CFG. Another approach is to represent alias sets (or points-to sets) symbolically, e.g., using BDDs (Lam et al. 2005). Unfortunately, BDDs are slow for flow-sensitive analyses, because of the large number of strong updates to pointer information (Hardekopf and Lin 2009). Hardekopf et al. overcome this in a partially symbolic, semi-sparse context-insensitive pointer analysis for C (Hardekopf and Lin 2009). Extending and evaluating those ideas in the setting of dynamic languages is a direction for future work.

References

L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

C. Anderson, P. Giannini, and S. Drossopoulou. Towards type inference for JavaScript. In *Proc. of the 19th European Conf. on Object-Oriented Programming*, pages 428–452, 2005.

D. F. Bacon and P. F. Sweeney. Fast static analysis of C++ virtual function calls. In *Proc. of the 1996 ACM SIGPLAN Conf. on*

Object-Oriented Programming Systems, Languages, and Applications, pages 324–341, 1996.

D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: composing static and dynamic analysis to validate sanitization in web applications. In *Proc. of the 2008 IEEE Symp. on Security and Privacy*, pages 387–401, 2008.

M. Bravenboer and Y. Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proc. of the 24th Annual ACM SIGPLAN Conf. on Object Oriented Programming, Systems, Languages, and Applications*, pages 243–262, 2009.

B. Cannon. *Localized Type Inference of Atomic Types in Python*. PhD thesis, California Polytechnic State University, 2005.

J.D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Proc. of the 20th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 232–245, 1993.

A. Diwan, K.S. McKinley, and J.E.B. Moss. Type-based alias analysis. In *Proc. of the 1998 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 106–117, 1998.

M. Emami, R. Ghiya, and L.J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proc. of the 1994 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 242–256, 1994.

M. Fähndrich, J. Rehof, and M. Das. Scalable context-sensitive flow analysis using instantiation constraints. In *Proc. of the 2000 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 253–263, 2000.

J.S. Foster, M. Fähndrich, and A. Aiken. Polymorphic versus monomorphic flow-insensitive points-to analysis for C. In *Proc. of the 7th Intl. Symp. on Static Analysis*, pages 175–198, 2000.

M. Furr, J. D. An, J. S. Foster, and M. W. Hicks. Static type inference for Ruby. In *Proc. of the 2009 ACM Symp. on Applied Computing*, pages 1859–1866, 2009.

M. Gorbovitski, K.T. Tekle, T. Rothamel, S.D. Stoller, and Y.A. Liu. Analysis and transformations for efficient query-based debugging. In *Proc. of the 8th IEEE Intl. Working Conf. on Source Code Analysis and Manipulation*, pages 174–183, 2008.

D. Goyal. Transformational derivation of an improved alias analysis algorithm. *Higher-Order and Symbolic Computation*, 18 (1-2):15–49, 2005.

A. Guha, S. Krishnamurthi, and T. Jim. Using static analysis for Ajax intrusion detection. In *Proc. of the 18th Intl. Conf. on World Wide Web*, pages 561–570, 2009.

S. Z. Guyer and C Lin. Error checking with client-driven pointer analysis. *Science of Computer Programming*, 58(1-2):83–114, 2005.

B. Hardekopf and C. Lin. Semi-sparse flow-sensitive pointer analysis. In *Proc. of the 36th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 226–238, 2009.

R. Hasti and S. Horwitz. Using static single assignment form to improve flow-insensitive pointer analysis. In *Proc. of the ACM SIGPLAN 1998 Conf. on Programming Language Design and Implementation*, page 105, 1998.

M. Hind. Pointer analysis: haven't we solved this problem yet? In *Proc. of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 54–61, 2001.

M. Hind and A. Pioli. Evaluating the effectiveness of pointer alias analyses. *Science of Computer Programming*, 39(1):31–55, 2001.

D. Jang and K.M. Choe. Points-to analysis for JavaScript. In *Proceedings of the 2009 ACM Symp. on Applied Computing*, pages 1930–1937, 2009.

N. Jovanovic, C. Kruegel, and E. Kirda. Precise alias analysis for static detection of web application vulnerabilities. In *Proc. of the 2006 ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, pages 27–36, 2006.

M.S. Lam, J. Whaley, V.B. Livshits, M.C. Martin, D. Avots, M. Carbin, and C. Unkel. Context-sensitive program analysis as database queries. In *Proc. of the 24th ACM SIGMOD-SIGACT-SIGART Symp. on Principles of Database Systems*, pages 1–12, 2005.

C. Lattner, A. Lenharth, and V. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *Proc. of the 2007 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 278–289, 2007.

O. Lhoták and L. Hendren. Context-sensitive points-to analysis: is it worth it? In *Proc. of the 15th Intl. Conf. on Compiler Construction*, pages 47–64, 2006.

O. Lhoták and L.J. Hendren. Scaling Java points-to analysis using spark. In *Proc. of 12th Intl. Conf. on Compiler Construction*, pages 153–169, 2003.

Y.A. Liu, S.D. Stoller, M. Gorbovitski, T. Rothamel, and Y.E. Liu. Incrementalization across object abstraction. In *Proc. of the 20th Annual ACM SIGPLAN Conf. on Object Oriented Programming, Systems, Languages, and Applications*, pages 473–486, 2005.

Y.A. Liu, M. Gorbovitski, and S.D. Stoller. A language and framework for invariant-driven transformations. In *Proc. of the 8th Intl. Conf. on Generative Programming and Component Engineering*, pages 55–64, 2009.

A. Milanova, A. Rountev, and B.G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Trans. on Software Engineering and Methodology*, 14(1):1–41, 2005.

M. Mock, D.C. Atkinson, C. Chambers, and S.J. Eggers. Improving program slicing with dynamic points-to data. *ACM SIGSOFT Software Engineering Notes*, 27(6):71–80, 2002.

G. Ramalingam. The undecidability of aliasing. *ACM Trans. on Programming Languages and Systems*, 16(5):1467–1471, 1994.

A. Rigo. Representation-based just-in-time specialization and the Psyco prototype for Python. In *Proc. of the 2004 ACM SIGPLAN Symp. on Partial Evaluation and Semantics-Based Program Manipulation*, pages 15–26, 2004.

E. Ruf. Context-insensitive alias analysis reconsidered. In *Proc. of the 1995 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 13–22, 1995.

M. Salib. Faster than C: Static type inference with Starkiller. In *Proc. of PyCon 04*, pages 2–26, 2004.

O. Shivers. Control-flow analysis in Scheme. In *Proc. of the SIGPLAN 1988 Conf. on Programming Language Design and Implementation*, pages 164–174, 1988.

V.C. Sreedhar, M. Burke, and J.D. Choi. A framework for interprocedural optimization in the presence of dynamic class loading. In *Proc. of the 2000 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 196–207, 2000.

M. Sridharan and R. Bodik. Refinement-based context-sensitive points-to analysis for Java. In *Proc. of the 2006 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 387–400, 2006.

B. Steensgaard. Points-to analysis in almost linear time. In *Proc. of the 23rd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 32–41, 1996.

J. Vitek, R. N. Horspool, and J. S. Uhl. Compile-time analysis of object-oriented programs. In *Proc. of the 4th Intl. Conf. on Compiler Construction*, pages 236–250, 1992.