Analysis and Transformations for Efficient Query-based Debugging *

Michael Gorbovitski K. Tuncay Tekle Tom Rothamel Scott D. Stoller Yanhong A. Liu
Computer Science Dept., State Univ. of New York at Stony Brook, Stony Brook, NY 11794
{mickg,tuncay,rothamel,stoller,liu}@cs.sunysb.edu

Abstract

This paper describes a framework that supports powerful queries in debugging tools, and describes in particular the transformations, alias analysis, and type analysis used to make the queries efficient. The framework allows queries over the states of all objects at any point in the execution as well as over the history of states. The transformations are based on incrementally maintaining the results of expensive queries studied in previous work. The alias analysis extends the flow-sensitive intraprocedural analysis to an efficient flow-sensitive interprocedural analysis for an object-oriented language with also a form of context sensitivity. We also show the power of the framework and the effectiveness of the analyses through case studies and experiments with XML DOM tree transformations, an FTP client, and others. We were able to easily determine the sources of all injected bugs, and we also found an actual bug in the case study on the FTP client.

1. Introduction

Debugging is the process of determining the source of an error given the symptoms of the error. While it is about program executions on particular inputs, it is necessarily also a process that requires significant effort analyzing the source code and often manipulating the code, manually, even with the help of good debugging tools. Methods and tools that can help reduce the effort needed are greatly desired.

Query-based debugging is a framework that allows powerful queries to be used in debugging. Unlike techniques that allow only values in a single scope to be used, it allows the use of all values in the program state, and even in the history of states. The results of these queries are used to watch conditions and trigger actions as the program executes. Although powerful queries can help make debugging much easier, they are also much more expensive to compute, and the values that the queries depend on change continuously as the program executes. These powerful queries are far from being supported in debugging tools, because of the significant overhead in computing the query results from scratch and the sheer difficulty in manually writing code that computes the query results incrementally as the program executes.

This paper describes a framework that allows powerful queries to be used in debugging tools, and describes in particular the transformations, alias analysis, and type analysis used to make the queries efficient. The framework allows queries over the states of all objects at any point in the execution as well as over the history of states. The transformations are based on incrementally maintaining the results of expensive queries studied in previous work. The alias analysis extends the algorithm for flow-sensitive, context-insensitive, intraprocedural analysis in [12] to an efficient flow-sensitive interprocedural analysis for an object-oriented language, with limited context sensitivity. The type analysis uses iterative computation of abstract data values, yielding more precise analysis results than standard type analysis while keeping the analysis efficient. Both analyses are critical in detecting precise changes where incremental maintenance of query results should be performed.

We also describe an implementation and experiments that show the power of the framework and the effectiveness of the alias analysis and type analysis. Case studies in the experiments include finding when certain properties of XML DOM representations are violated, determining sources of out of bounds exceptions for array indices, and finding out-of-order commands sent by an FTP client. We were able to easily determine the sources of all injected bugs, and we also found a non-injected bug in the FTP client.

Query-based debugging has been studied for at least a decade [19] and has received increased attention in recent years [18, 24, 22, 30], and much other related work has also been done, as discussed in Section 6. To the best of our knowledge, no previous work supports the general forms of queries that our framework allows and achieves the level of efficiency that our methods do. Compared with our own work on incrementalization and run-time invariant checking [20, 11], this paper for the first time studies debugging and describes our alias analysis and type analysis precisely, and supports their efficiency as part of the experiments. Our

*This work was supported by NSF under grants CCF-0613913, CNS-0509230, and CCR-0306399, and by ONR under grant N00014-07-1-0928.

flow-sensitive alias analysis for an object-oriented language extends prior work [12].

2. Framework

The premise of query-based debugging is that allowing users to easily write expressive queries about the program execution helps them find and diagnose bugs.

In this section, we describe the *query language*, its features, and three classes of errors, as well as queries that help to find the bugs that cause the errors. Then, we discuss the efficient implementation of the language.

Debugging rules and queries. The general form of a debugging rule is shown in Figure 1.

foreach(*query*) :
  action
(**de in** *scope* (*field_decl*|*method_decl*)?)*
(**at** *update*
  (**if** *condition*)?
  (**de** (**in** *scope* (*field*|*method*)+)*)?
  do (**before** *maint* (**after** *maint*)?) |
       (**instead** *maint*)
)*

Figure 1: General form of a debugging rule.

A *query* has the form (v_1 `in` S_1, ..., v_k `in` S_k: *condition*); *condition* is a conjunction where each conjunct has the form e_1 *op* e_2, *op* is `==`, `!=`, `in`, or `not in`, e_i being `v` or `v.f`, with `v` a variable and `f` a field; or a boolean expression whose value depends only the objects in the containers iterated over by the query (S_1, ..., S_k), the fields of these objects, and any immutable objects. The set of tuples of values of v_1, ..., v_k, such that *condition* holds is called the *query result*. *action* is a sequence of statements to be executed for each tuple in the query result.

The rest of the syntax is based on InvTS [20]. The `at` clause contains a code pattern *update* that gets matched against expressions in the program. The *update* clause may contain Python code and meta-variables. Meta-variables are denoted by prefixing their name with "`$`", and they match expressions in the program. For each part of the code in the subject program that matches the pattern in the `at` clause, if the *condition* in the `if` clause is satisfied, then the declarations in the `de` clause are inserted into the program in the specified scope, and the *maint* code in the `do` clause is inserted `before`, `after`, or `instead` of (i.e. replace) the code matched by *update*. In the `if` clause, the condition is built from standard logical connectives and functions defined for the subject language. For example, `class`($expr$) returns the class in which $expr$ appears, and `type`($expr$) returns the type of $expr$. In the `de` clause, *scope* can be `global` or the name of a class, method, package, or file.

Violation of invariants. Detecting violations of data structure invariants as soon as they occur, instead of waiting until incorrect output is produced, can make it much easier to find and diagnose bugs.

For example, tree data structures in the Python XML DOM implementation have the invariant that when node `a` is in the `children` set of node `b`, `a.parent` must refer to `b`. Finding the node that the child was added to using standard debugging techniques is difficult, due in part to aliasing. For example, the following code aliases `x` to `parent.children`, and then updates the set through `x`, without accessing the `children` field:

```
x=parent.children
x.add(child)
```

The debugging rule in Figure 2 says to stop the DOM implementation when a child and parent are inconsistent. For simplicity, consistency is checked at every program point. To check consistency only at specified program points (e.g., method return points), we could extend the *action* with an `if` statement that checks whether we are at such a point.

```
foreach (n in extent(Node),
         m in extent(Node) :
         m in n.children and
         m.parent != n ):
   report("Child ", m, "is a child of ",
     n, ", but ", n, " is not the",
     " parent of ",n)
   stop()
```

Figure 2: A child's `parent` field must point to its parent.

For every class T, `extent(`T`)` is a special set defined by our framework to contain all currently existing objects of type *T*. `report` and `stop` are functions in the subject programming language: `report` takes any number of arguments and prints the concatenation of their string representations; `stop` stops the program and drops into a debugger. The *condition* in this case is `m in n.children and m.parent != n`. It is a conjunction of relational joins (because each conjunct contains multiple variables).

It is easy to write this rule in our framework, but it is difficult to manually write code that would compute the value of the query efficiently for the following reasons: (1) The result of the query may be changed by any statement that adds an object to a collection, such as : `x=o.children; ...; x.add(o)`. It is tedious and error-prone to write code to intercept all calls to `add` and determine whether the target object equals the `children` field of some instance of `Node`. (2) Efficiently maintaining the result of a join over two changing sets is non-trivial, and involves the maintenance of additional information, etc. In our framework, the user writes the rule, and our system does the rest, generating correct and efficient code for it and inserting that code properly in the program to be debugged.

Violations of temporal properties. Bugs often manifest themselves as violations of temporal properties. Detecting

```
foreach (c1 in $exec_commands,
         c2 in $exec_commands :
         c1.cmd == 'ls' and
         c2.cmd == 'cwd' and
                 c1.host == c2.host
        ):
   report('ls and cwd being executed',
          ' at the same time.!')
   stop()
de in global:
  $exec_commands=set()
at $x.cwd($dir):
if type($x) == ftplib.FTP:
do before:
   $c=command($x,'cwd')
   $exec_commands.add($c)
do after:
   $exec_commands.remove($c)
at $x.list():
if type($x) == ftplib.FTP:
do before:
   $c=command($x,'ls')
...
```

Figure 3: A rule that makes sure no new FTP `ls` commands are sent while there are outstanding `cwd` commands.

these violations immediately, which may be well before incorrect output is visible, can make it much easier to pinpoint the source of the error. Our framework allows users to write queries that express temporal properties using debugging rules that transform the program to maintain information about past events. This is similar to aspect-oriented programming [16]. We illustrate such a query with a case study involving nftp, an FTP synchronization tool.

Nftp did not copy some directories that it should copy. Inspection of the logs on the FTP server reveals that after changing directories, nftp is trying to copy files from the old directory, not the one it changed into. Since the nftp is multi threaded, we guess it does not wait until the `cwd` command completes before enumerating the files and starting to copy them. This bug is not obvious from inspection of the nftp code, because the commands appear in the correct order in the code; to realize the error, one needs to think about the use of multiple threads and how they are synchronized. It is also difficult to verify this hypothesis using standard debugging techniques, as there is no easy way to find out to which commands the tool has not yet received a reply to, as the `ftplib` module that is used by nftp does not create an object per sent command and does not internally maintain the set of outstanding commands.

The rule in Figure 3 stops the program when a new `ls` command is sent to a host while a `cwd` command to that host is still outstanding. The rule maintains (and queries) `$exec_commands`, a set of outstanding FTP commands. At all places in the program where the `cwd` command is executed by an `ftplib.FTP` object (`at` and `if` clauses), it is added to `$exec_commands` immediately beforehand (`do before`). It is removed from the set immediately after (`do after`) the `cwd` call returns. The same is done for `list` and other FTP commands. The dollar sign indicates that `$exec_commands` is a meta-variable; it will be instantiated with a fresh program variable, whose value will be set to a new empty set (`de`). `command` is a class we define, with fields `cmd` and `host` to store the command and the host nftp is connected to, respectively.

Causes of uncaught exceptions. Many bugs manifest themselves as uncaught exceptions. For example, in Python, an expression `$L[$R]` throws an `IndexError` if the index `$R` is out of bounds for the ordered collection (e.g., a list) `$L`. To debug such an error, the user would like to know which assignment led to it. The query in Figure 4 finds the earliest update after which the error became "inevitable", i.e., `$L[$R]` would still throw `IndexError` after every subsequent update to `$L` or `$R`. This is difficult to do with standard debugging techniques for two reasons: we do not know the list object involved in the `IndexError` until it occurs, and there might be multiple ways to update the index if the index is inside an object, via aliasing of that

```
foreach (c in $C, i in $I :
         c.value != None and
         i.value != None) :
   if outOfRange(c.value, i.value):
      if (c.value,i.locId) not in $bad:
         $bad[c.value,i.locId]=$LOCATION
   else:
      if (c.value,i.locId) in $bad:
         del $bad[c.value,i.locId]
de in global:
   $bad={}
   $C=set()
   $I=set()
   def wrapper(L,R,locIdR):
      try: return L[R]
      except IndexError, error:
        report ("Became inevitable at: ",
               bad[L,locIdR])
        stop()
var $L, $R
at $L[$R]:
if line(12) and file('t.py'):
do instead:
   wrapper($L,$R,locId('$R'))
at $e:
if part($e,'$x','alias($x,$L)
         and update($x)'):
do before:
   $obj=Update(locId=locId('$x'),value=$x)
   $C.discard($obj)
   $C.add($obj)
```

Figure 4: A rule that helps determine the cause of an uncaught exception.

object. After determining that the exception occurs at line 12 in the file `t.py`, the program needs to be executed again, after instrumentation with this query, to find the updates.

The rule works by replacing `$L[$R]` with a function call that returns the result of the lookup if successful; otherwise it prints the location at which the `IndexError` became inevitable. To accomplish this, the rule uses a query to maintain a map `$bad` from objects which may be aliased to `$L`, and variables (e.g. fields) that could be aliased to `$R` to locations after which the error becomes inevitable. The query is over `$C` (Collection) and `$I` (Index), sets that contain the last place where variables and objects that `$L[$R]` depends on were last updated. Computing `bad` using a query allows us to write what `bad` is declaratively, instead of manually incrementally computing changes to it whenever `$C` or `$I` are updated. `outOfRange(a,b)` returns whether `a[b]` will throw an exception. `$LOCATION` is a keyword that expands to an object that identifies the statement being transformed. `Update` stores two fields: `locId` and `value` store an object identifying an instance of variable, and an arbitrary value, respectively. Only `locId` is used for comparing instances of `ID`. Thus, `$C.discard($obj)` removes the entry with the same `locId` as `$obj` from `$C`, and `$C.add($obj)` adds to it `$obj` with the new `value`. `locId` is a function that generates an identifier that uniquely identifies an instance of an lvalue. Updates to `$R` are handled in the same way as updates to `$L`, under the substitutions `$L`⇒`$R`, `$C`⇒`$I` (This part of the rule is not shown). `part($e,'$var',cond)` is a special function that, for each subexpression of `$e`, evaluates whether `cond` is true for that subexpression, and binds `$var` to it. Note that if there are no updates to either `$L` or `$R`, then code to maintain `$C`, etc. is not inserted.

This rule can be reused by changing `line(12) and file('t.py')` to indicate the file and line at which the `IndexError` occurred. Similar rules can identify causes of other kinds of exceptions and other invariants.

Implementation. The straightforward way to implement this language is to evaluate every query at every program point. This is very inefficient, especially if the size of the collections queried over is large. Evaluating each query only at program points that affect its result is more efficient, yet still requires repeated reevaluation of the query. For all queries specified by the programmer, our implementation incrementally maintains their results whenever a set or object the queries depend on changes. The transformations used to achieve this are described in [11]. There are two steps involved in this approach: (1) generating maintenance code, and, (2) applying the maintenance code at the appropriate places.

In step 1, we generate maintenance code that properly maintains the query results in the face of all possible updates to the data the query depends on. This is accomplished by compiling the query into an InvTS rule [20], which then transforms the subject program so that it incrementally maintains the query result. The resulting rule looks like a rule in Figure 1, except that it only consists of `at`, `if`, `de`, and `do` clauses, and says "`at` a given update `if` a condition holds `do` maintenance code". InvTS (the Invariant-driven Transformation System) is a program transformation system geared towards source-to-source transformations that maintain invariants.

In step 2, we apply the maintenance code at all places where the query result might change. This involves determining all locations that update the variables the query might depend on. InvTS uses control-flow, data-flow, type, and alias information to determine which updates do not affect the query result, eliminating the need to insert maintenance code guarded by runtime checks (of aliasing, etc.) at such updates. Also, it is often possible to statically evaluate the `if` clauses, especially if the condition consists of only comparisons of type expressions. This has the effect of reducing the number of needed runtime checks, thus reducing the overhead of maintaining the query result, as shown in Section 5 (especially Figure 5). The next two sections describe our alias analysis and type analysis. We call the system we have implemented qbdPy (Query-based debugging for Python).

3. Alias analysis

We use alias analysis to reduce the number of runtime checks, as an update to a variable that is not aliased to anything in the query cannot affect the query result. Clearly, more precise alias analysis allows more runtime checks to be eliminated.

Two variables in a program are *aliased to each other* if they refer to the same location or object. *Alias analysis* of a program computes pairs of variables that are aliased to each other. Computing these pairs precisely is undecidable [25], therefore only an over-approximation, called *may-alias analysis*, is safely used for our purpose. A may-alias analysis computes pairs of variables that *may* be aliased to each other.

A may-alias analysis is flow sensitive if it computes pairs at each program point. It is interprocedural if it propagates these pairs (and changes to them) through procedures (for example, by analyzing an inter-procedural control flow graph). The analysis is context insensitive if it can not distinguish different calls of the same procedure. Note that flow sensitivity, context sensitivity, and the analysis being inter vs. intraprocedural are orthogonal [17]. During the development of InvTS, we have investigated flow-insensitive [3] and flow-sensitive but context-insensitive may-alias analysis algorithms. We found their precision to be insufficient. We settled on the current flow-sensitive, partially context-sensitive algorithm as sufficiently precise.

An intraprocedural, flow-sensitive, and context-

insensitive may-alias analysis has been described in [14], and an optimally efficient algorithm for it is given by Goyal [12]. However, this analysis is intraprocedural and does not handle classes. To analyze an object-oriented language such as Python or Java, the analysis needs to be extended to handle interprocedural analysis and classes. We describe how we handle these differences and give a time complexity analysis compared with the algorithm in [12]. We did not perform fully context-sensitive analysis because it is much more expensive, and our analysis is sufficiently precise for our purposes.

Interprocedural analysis. Extending the intraprocedural analysis to an interprocedural one requires handling the following features: function calls and function parameters. We rename variables such that same named variables are named differently if their scopes are different. Then we build the control-flow graph except for the function calls and parameters, and modify the control-flow graph as follows:

For each parameter p of a function and its argument a at a call to p, we create a control flow graph node n for a new statement $p = a$, add an edge from the function call node to n, and add an edge from n to the entry point of the function.

For each function call appearing on the right hand side of an assignment, we replace the call with a fresh variable v; for each return statement `return` r, we create a node for a new statement $v = r$, add an edge from the return statement to the new statement, and consider the new statement as an exit point of the function. We then add edges from all exit points of the function to the statement that follows the call.

Classes. To add classes to the analysis, we handle constructors, member function calls, and field accesses. We flatten classes in the sense that we remove class definitions and turn member functions into regular functions and ensure that they take the object that they are invoked on as a parameter (`this` in Java, `self` in Python). We rename all variables and functions to reflect their enclosing classes. For each field f of an object o, we treat $o.f$ as a possible name for a variable and handle it as described below.

For each reference to an object in the program we proceed as follows. If we encounter an object construction, `o = c(p1,p2,..)`, in order to maintain a persistent reference to a created object, we replace the statement with two statements: `ref_o = c(p1,p2,..); o = ref_o`, where `ref_o` always refers to the object created at this program point. For each constructor and member function, we build the control-flow graph as described.

To handle fields, at a call to a member function on an object o, if o is an object of class c, then for each field f of c, we create a node n for a new statement *self*.$f = o.f$, add an edge from the function call node to node n, and add an edge from n to the entry point of the function. We also create another node m for a new statement $o.f =$ *self*.f, add an edge from each exit point of the called member function to m, and add an edge from m_i to the next statement at the call site. The $self$ object is handled in the same way, by introducing the statement $self = o$ before the function.

Other features. We can also handle languages that allow nested function declarations such as Python, where a function f nested inside another function g can read the variables of g that are in scope at the declaration of f. We take each such function declaration to the global scope, add the variables in the local scope as parameters to f, and also add those variables as arguments to the calls of f. Similar modifications combined with handling of classes allow handling of nested classes as in Java.

For languages that allow polymorphism, at a call to a polymorphic method, we add edges from the call to all possible methods. We reduce the set of possible methods by type analysis.

By similar extensions, the methods described above can be used to perform may-alias analysis on any object-oriented programming language.

Optimizations. For a language that does not allow arbitrary type-casting such as Python, we utilize the type system described in this paper, and disallow the addition of pairs of variables which are of incompatible type to the analysis.

Other optimizations based on static analysis are also possible. In particular, if the value of the condition of an *if* statement can be determined statically (this is often possible if the condition only involves $const_{...}$ types) then we eliminate the dead branch.

Summary and time complexity. By handling the above discussed features and then using Goyal's algorithm, we obtain a flow sensitive, interprocedural may-alias analysis for an object-oriented language.

Goyal [12] gives an $O(N \times V^2)$ algorithm for the intraprocedural analysis, where N is the size of the program, and V is the number of variables, so it is bounded by $O(N^3)$. With the extensions above, assuming that the arity of functions and the number of fields per class are bounded by constants, we obtain an $O(N \times V^2)$ algorithm for the extended analysis where N is the size of the program and V is the number of original variables. Note that this is optimal, as Goyal's algorithm is, because the output size is bounded by $O(N \times V^2)$

Extensions. The analysis discussed so far is context insensitive. A form of context sensitivity can be added to make the analysis more precise but keep the analysis result to be one set of alias pairs per program point: after variable renaming, make a copy of the function for each call to the function, but do not rename local variables to be distinct again; do everything else as before; and finally union results from copies for each function. An optimization for space is to not make the copy until the call is analyzed and not to keep the copy after it is analyzed. The code size becomes $O(N^2)$ but the number of variables stays the same,

so the time complexity is $O(N^2 \times V^2)$, which is bounded by $O(N^4)$. This is the algorithm we implemented.

Another possible way of adding of context sensitivity is by inlining non-recursive functions. This makes the analysis more precise by limiting propagation of may-aliases that are not possible in the calling contexts. If each function is of size $O(s)$, there are $O(c)$ calls to each function, and the depth of calls for non-recursive functions is $O(d)$, then the analysis now takes $O((N + (s \times c)^d)^3)$. If one assumes that the depth of calls and the number of calls to functions are bounded by constants, then this analysis is $O(N^3)$.

4. Type analysis

Our system uses static type analysis to reduce the number of runtime checks. If a variable is being updated, and variables (or fields) of the same type are not used in the query, then the update cannot affect the result of the query, and the corresponding runtime check can be eliminated.

The goal of our type analysis for Python is not to statically ensure type safety, but to obtain type information that can be used in various analyses, such as may-alias analysis. Thus, the type system for Python we propose collects as much information in its types as possible. For example, if a variable `v` can only evaluate to `1` or `2`, we infer that `v` has the type $union(int_{const}(1), int_{const}(2))$.

First, we present a type system for Python, and a type inference algorithm for it. Then, we give an extension for the type system and show the necessary changes to the inference algorithm for this extension.

Basic type system. A Python expression's value, evaluated at runtime, can be put into one of the following groups: int, $float$, $boolean$, $string$, $list$, $tuple$, set, $dict$ (a map), $class$, $function$, $instance$ (an instance of a class), $method_{bound}$, $module$ (similar to a package in Java), or $none$. We make each of these groups a type.

A Python expression can evaluate to values of different types each time it is evaluated. To accommodate this, we introduce the union type $union(type_1, \cdots, type_k)$, where each $type$ is any type other than a union type. The type inference rules define a multimap at each program point from expressions to their possible types. Most of the rules mirror Python semantics in a straightforward way. For example, a rule for handling addition of two ints is

$$\frac{z = x + y, \quad int \in in[x], int \in in[y]}{out = in \cup \{z \rightarrow int\}}$$

x, y, and z are expressions in the program, in is the multimap from expressions in scope to their types right before the current program point. out is the multimap from expressions in scope to their types right after the current program point. Other rules are similarly straightforward. Encoding Python semantics into rules resulted in 67 rules. We do not need rules for union types, because they are represented implicitly by using a multimap to allow an expression to have multiple types.

Type inference. We implement an algorithm based on iterative type inference in order to infer the multimap out for every node in the control flow graph (CFG). The algorithm starts by converting the CFG to three address code (3AC) and assigning types to every literal constant in the program. Then, the least fixed-point of the typing rules is computed. For a node m, in_m, a multimap containing the types of expressions that are in scope at m, is computed: $in_m = \bigcup_{n \in pred(m)} inscope_m(out_n)$, where $inscope_m(out_n)$ contains the types of expressions in out_n that are in scope at m, and $pred(m)$ is the set of predecessors of m in the CFG. The sizes of in_m and out_m are limited to the number of expressions in scope at m, called S. out_m is computed by evaluating the typing rules under the substitution $\{in \rightarrow in_m, out \rightarrow out_m\}$. The output of the algorithm is $O(N)$ multimaps, with at most $O(S)$ entries per map, where N is the size of the CFG. An incremental workset algorithm that incrementally maintains out for each node is used to compute the least-fixed point in $O(N \times S)$ time.

Extended type system. We extend the basic type system to distinguish different integers, different classes, different functions, etc. For all the basic types presented above except $none$, we introduce some subtypes. We define type T to be a subtype of T', denoted as $T \prec T'$, if $values(T) \subset values(T')$, $values(T)$ defined as the set of Python values that have type T.

For int, we create the following subtypes $int_{const}(x)$, $int_{range}(from, to)$ and int_{non_neg}, with the obvious meaning; x, $from$, and to denote integer constants. We introduce similar subtypes for $bool$ and $float$. For $string$, we differentiate between strings of known content ($string_{const}(x)$) and known length ($string_{fixed\_length}(n)$). For $list$s, we introduce subtypes that represent lists of known content, lists of known length, and lists of unknown length but known homogeneous type. The same is done for $tuple$, set, and $dict$. $module$ has two subtypes. The first is when the module name and all the module variables are known ($module_{const}(name, (v_1 : t_1, v_2 : t_2, \dots))$, $name$ is the given name of the module, $(v_1 : t_1, v_2 : t_2, \dots)$ is the set of $variable$:$type$ pairs that represent the bindings exported by the module); the second, where just the module name is known ($module_{known\_name}(name)$). $func$tions whose bodies are known are represented by the subtype $func_{const}(t_{return}, (t_{p1}, t_{p2}, \dots), [name])$, with t_{return} being the return type, t_{p1} being the type of the first parameter, and $name$ being the optional (hence in []) name of the function. A similar encoding to $func_{const}$ is used for bound methods ($method_{bound}$). For brevity, we omit the precise types of the parameters, such as a boolean for $bool_{const}$, and a floating-point number for $float_{const}$.

Extended type inference. Most of the new typing rules are straightforward. For example,

$$\frac{z = x + y, \quad int_{const}(p) \in in[x], int_{range}(f,t) \in in[y]}{out = in \cup \{z \rightarrow int_{range}(f+p, t+p)\}}$$

The extended type system contains 312 rules. Many rules are needed to capture the semantics of builtin functions, such as `range`, a function that takes one integer p, and returns a list containing integers 0 to $p-1$, in order. One of the rules for `range` is:

$$\frac{z = range(x), \quad int_{const}(p) \in in[x]}{out = in \cup \{z \rightarrow list_{const}((int_{const}(i) \mid i \in 0..p-1))\}}$$

Note that the list comprehension $(int_{const}(i) \mid i \in 0..p-1)$ gets evaluated to $list_{const}$ during type inference. Similar rules exist for handling list and set operations.

The following rule handles a two-parameter function call. It iterates over all possible types of parameters (the $\forall A \in in[a], \cdots$) , matching it to existing $func_{const}$ signatures for f, and adding all possible types of the return value to out.

$$z = f(a,b), \qquad out = \{\} \ ,$$
$$\forall A \in in[a], B \in in[b] \ : \quad \frac{func_{const}(t_r, (A,B)) \in in[f]}{out = out \cup in \cup \{z \rightarrow t_r\}}$$

We extend the type inference algorithm for the basic type system to infer extended types. Changes are needed for two reasons. (1) The basic algorithm does not terminate because there are now an infinite number of types ($int_{const}(1)$, $int_{const}(2)$, etc.). (2) The algorithm can be made more precise by computing an out multimap per outgoing edge of each node.

We call the process of adding another type to an expression due to a type judgement *type extension*. The type complexity of a $union$ type is the sum of the type complexities of its members. The complexity of a type with type parameters (such as $list$) is one more than the sum of the complexities of its type arguments. The complexity of other types is 1, except for int_{range}, whose complexity is one more than the number of times that the range of values has been extended. To ensure termination, we introduce a cutoff c, and whenever the type of an expression e has $complexity > c$, we apply *generalization rules* that either replace two or more of e's types with a single supertype of the replaced types (e.g., $int_{const}(x), int_{const}(y) \Rightarrow int_{range}(min(x,y), max(x,y))$) or replace one of e's extended types with an unparameterized basic type (e.g., $int_{range}(min(x,y), max(x,y)) \Rightarrow int$). The following sequences of types represent (in a simplified way) other generalization rules in our system: $int_{const} \Rightarrow int_{range} \Rightarrow int_{non\_neg} \Rightarrow int$, $list_{const} \Rightarrow list_{homogeneous} \Rightarrow list$, and $list_{const} \Rightarrow list_{unknown} \Rightarrow list$. When more than one generalization rule applies, we use a heuristic to prioritize them. As an optimization, every time we run generalization on a node replacing some of its types with a supertype T, we also delete all of its types that are subtypes of T. This removes redundant entries from in_e and out_e.

To make type inference more precise, we modify it to store distinct outs for each edge coming out of a node. To see how this can help, consider the following code:

```
if x == 1:  ...
else:  ...
```

The `if` statement forms a CFG node n_{if} with two outgoing edges. If $in_{n_{if}}$ contains the entry $x \rightarrow int_{const}(1)$, the basic inference algorithm must include this entry in $out_{n_{if}}$, while the extended inference algorithm can omit it from the out multimap on n_{if}'s outgoing edge to the false branch. A similar approach can be used to split the out of `for` and `while` loops. `if`, `for`, and `while` nodes have $O(1)$ outgoing edges, so the number of multimaps produced is still $O(N)$. Other optimizations include the sharing of type signatures between multiple in and out maps when the signatures are the same (a variation of copy-on-write).

Extending our analysis of the running time to the extended type inference algorithm, we note that an expression may not have more than c types, and that an expression can not be generalized more than g times, where g is the height of (i.e., length of the longest chain in) the subtype relation with all int_{range} to int_{range} relationships of length greater than c truncated to length c. Thus, the total number of distinct types that an expression may assume is $O(g \times c)$. Thus, the size of each out multimap is still $O(S)$, and the worst-case running time is still $O(N \times S)$. S typically does not grow much with program size. This is experimentally verified in Section 5, where the running time shows linear behaviour in N.

5. Experiments on overhead

Overhead of debugging has two components: the slowdown incurred due to running the program in qbdPy, and the time it takes for qbdPy to instrument the program to be debugged. To show that our technique does not introduce excessive overhead, we perform two sets of experiments. The first set of experiments measures the slowdown due to the program running in qbdPy; the second measures the time to instrument the program and also contains experiments that verify the running times of analyses derived in sections 3 and 4.

All experiments were performed on Windows Vista, running on a Core 2 Duo (Q6600@3.0GHz) machine with 8GB of memory, of which 6GB were free. For all examples, Python 2.5.1 was used.

5.1. Slowdown due to running program in qbdPy

We demonstrate that qbdPy does not introduce excessive slowdown due to the program running in it, by using qbdPy to find different bugs in programs from multiple domains: violations of data structure invariants in XML DOM

transformations, violation of specifications in an FTP client, and uncaught exceptions in an XML DOM transformation benchmark program due to injected bugs. For each program, we report the performance of the program outside of qbdPy; the program's performance in qbdPy when it uses incremental checking and maintenance; the program's performance when static analyses are individually disabled; and the program's performance when it does not use incremental checking and maintenance.

XML DOM transformations. For a program that uses an XML DOM tree to be correct, there are a number of properties that must not be violated for the tree to avoid bugs. Usually, such bugs will manifest themselves in a further stage in the program after a property has been violated. We take the lxml Python XML library, and, for its benchmark programs detect violations of the following properties of the XML DOM tree: (1) if an element is a child of another element, then its parent field must reference the element whose child it is; (2) no two elements may have the same element as their child, nor may an element have itself as a child. As the lxml benchmark code does not itself contain these bugs, we have injected the appropriate bug for each experiment.

Parent field must be valid. In an XML tree, all non-root nodes must have a valid parent field, i.e., element e has a child c iff c.`parent` is e. The rule in Figure 2 stops the program when an element that violates that property is found. Figure 5 shows that the overhead of running the incrementally instrumented program in qbdPy is 67%. It also shows that type and alias analysis decrease overhead from 109%-176% to 67%. In contrast, non-incremental instrumentation is quadratic in the number of elements alive in the program as it iterates over two extents of elements. The benchmark times out after 20 minutes with the non-incremental instrumentation, since it does $O(\#element^2)$ additional work per update, and the benchmarked document has 10 million XML elements.

No shared child and not own child. In an XML document, an element may be either a root, or a child of at most one element. Also, an element cannot be a child of itself. We omit the actual rule, as it is very similar to the previous rule. Figure 5 shows that the overhead of running the program in qbdPy, with all analyses enabled, is 85%. It also shows that type and alias analysis both provide a significant reduction of overhead, just like the previous example. The non-incremental version times out after 20 minutes because it iterates over three extents of elements, doing $O(\#element^3)$ extra operations per update, and the benchmarked document has 10 million XML elements.

These experiments show three things: query-based debugging that incrementally maintains its results can be efficient even for complex queries that involve multiple joins and membership tests. We also see that when joins used by the query have a high selectivity, as these do, the running time of the instrumented program is not very dependent on the query, but more so on the number of objects (and classes) for which we maintain extents. Finally, these experiments show that maintaining the query results non-incrementally is infeasible, as the experiments time out whenever query results are computed non-incrementally.

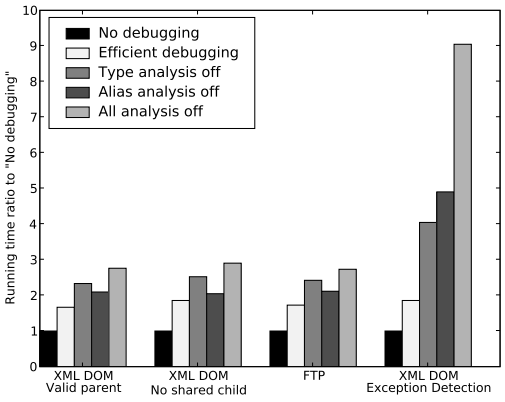

Figure 5: Running times of applications in qbdPy, normalized to the running time of the applications outside of qbdPy.

A Python FTP client. We found a bug in a program that downloads directories from multiple machines [15]. This bug involve directories being omitted from synchronization. The bug is due to the FTP client issuing commands before receiving the reply for those commands. The query in Figure 3 finds the location at which a command of `ls` is executed when a `cwd` is pending.

We ran the program with 10 threads, with 30 directories totaling 20GB over a 1GBit connection, ensuring that the program would be CPU bound. Figure 5 shows that the overhead introduced by the query is 73%. It also shows that type and alias analysis both provide a significant improvement, reducing overhead from 173% to 73%. The non-incremental version is considerably slower, as there are many threads running, and `$executing_commands` contains many elements. This accounts for it timing out after 20 minutes. Precise time taken by all versions of the program can be seen in Table 1.

The FTP client example shows that querying a complex program over a view that has to be created (i.e., in ways not assumed by the program's creator) is easily done with our framework by specifying complex program transformations, such as maintaining the set of outstanding commands.

Automated determination of causes of exceptions. In the final case study in Section 2, Figure 4, we presented a query that, given an IndexError caused by an expression of type `A[B]`, and the line and file it occurred on, will tell the programmer where all variables in the expression were modified when it became inevitable that the exception would occur. We injected a bug that would cause an Index-

	Running time					Instrumentation + running time	
	No debugging	All analysis	No type analysis	No alias analysis	No analysis	All Analysis	No analysis
lxml - Valid Parent	21s	35s	49s	44s	58s	70s	78s
lxml - No shared child & no self child	21s	39s	53s	43s	61s	77s	83s
nftp - Wait until commands complete	326s	563s	790s	690s	891s	594s	912s
lxml - Exception cause detection	21s	39s	85s	103s	190s	92s	215s

Table 1: Time taken for experiments under differing optimizations.

Error into lxml, and ran it after applying this debugging rule to it. The result, as can be seen in Figure 5 and Table 1, is that the slowdown incurred by such a query is 85%, which is surprisingly low given the low selectivity of the join condition. The drastic increase of the overhead when type and alias analyses are turned off (from 85% to 805%), as seen from Figure 5, explains the high performance of the query.

5.2. Running time of the qbdPy

To verify that the running time of qbdPy instrumentation is not prohibitive, for each of the programs in the previous section we perform the following experiments: measure time taken by qbdPy instrumentation with all static analysis turned on, with all static analysis turned off, and with type and alias analysis turned on individually. Table 1 presents the results. None of the programs take longer than 1 minute to instrument with all analyses turned on, and none took less than 15 seconds with all analyses turned off. This indicates qbdPy is fast enough to be used as part of the edit-compile-debug cycle, and that all available analyses should be performed due to the great increase in runtime performance.

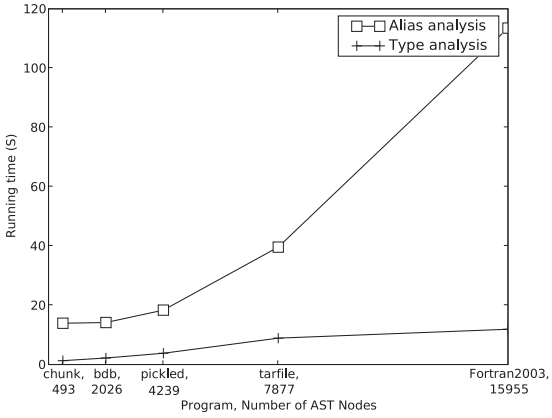

Figure 6: Running times of type and alias analysis.

Complexity of type and alias analysis. To verify the running time complexity of type and alias analysis, we measure the time of running type and alias analysis on a representative set of Python programs, with program sizes varying from 493 to 15955 CFG nodes. From Figure 6, we can see that the running time is quadratic for alias analysis and linear for type analysis. For type analysis, this is better than the theoretical worst-case $O(N \times S)$, and is in line with the expectation that S is typically a constant. For alias analysis, this is much better than the worst-case $O(N^4)$.

6. Related work

Query-based debugging has recently received a great deal of attention [18, 24, 22, 30], mostly in the form of query languages that query over a given program state. These languages allow one to specify an assertion for a bug, and then stop execution when the assertion holds. These systems primarily differ in the range of specification, and the time complexity. Our work was inspired by the work of Lencevicius et al. [18], and our language is an extension of the query-based debugging language of [18]. The method in [18] allows non-nested comprehensions over extents, with the condition being a join or a side-effect free function over a single variable of the comprehension, and recomputes the entire query whenever sets that the query depends on are updated. Our method avoids recomputing the query when the sets it depends on are updated, while increasing the expressive power of the allowed queries by including predicates over multiple variables and joins of membership tests. We also add features that allow arbitrary program transformations, e.g., to maintain history.

Potanin et al. [24] allows querying snapshots of object graphs, and also performs these queries non-incrementally. PQL [22] allows queries over past states of the program, but not over extents. It uses BDDs to efficiently compute the query results. PTQL/PARTIQLE [10] allows queries over sequences of past actions (such as variable assignment) of the program, but not over sets/extents in the program. It uses join ordering to efficiently evaluate these queries at run-time, but does so non-incrementally. JQL [30] extends Java to support both comprehensions and extents, with expressive power similar to our system, for introducing comprehensions as a first-class construct into Java, rather than debugging. Recent work on JQL [31] adds incremental maintenance of JQL queries for updates to the data they depend on. We support a larger set of conditions on queries: we can incrementally maintain query results for queries that contain a condition of the form `a in b.f`. The `at` and `de` clauses allow us to do program transformations that maintain data structures that would be unavailable to a query language, such as a set of outstanding FTP commands.

Aspect-oriented programming. An important feature of an aspect-oriented programming language is its lan-

guage for defining pointcuts. The pointcut language of AspectJ[16] is somewhat limited; other proposals [2, 27] are more expressive. In particular, these proposals allow advice to execute based on the history of program execution.

The goals of our system are similar to the goals of languages for specifying pointcuts. The similarities are between `at/do` clauses and pointcuts/advices. The differences come from the inability of AOP to derive how to maintain query results; reasonable performance requiring the currently lacking consideration of the same problems as our system (type and alias analysis of dynamic languages).

Alias and type analysis. Alias analysis has been studied extensively [14], in different flavors. Context and flow insensitive methods include Andersen's [3], Deutsch's [7], and Steensgaard's [28]. Context sensitive methods include Whaley [29] and Emamil [9]. Two flow sensitive methods are Choi et al.'s [4] and Lundberg and Lowe's [21], with running times of $O(N^7)$ and $O(N^4)$, respectively.

Intraprocedural flow-sensitive methods include Hind's [14], which takes $O(N^5)$ time, and Goyal's [12] method that takes time $O(N^3)$. We take Goyal's algorithm, make it interprocedural and slightly context-sensitive, and apply it to Python, where the running time of our method is $O(N^4)$. The method we use for the handling of classes is similar to [21]. Our extended method of handling function calls is similar to summaries with weak updates by Wilson et al.[29], with the distinction that instead of computing summaries and then using them in the analysis, at each call site, we analyze the function, and, after each analysis, incrementally maintain the function summary.

Type analysis originated from the work by Curry and Feys on simply typed lambda calculus [5]. Progress by Hindley, Milner, and Damas produced a type inference algorithm that supports polymorphic references [6]. This algorithm is the standard type inference algorithm for ML [6]. This was extended later by works of Plevak [23], Rémy [26], and Graver [13] to object-oriented languages, such as ML with OO extensions and a subset of Smalltalk. These works were based on representing the program as a set of constraints, and solving these constraints via unification and similar methods. The Cartesian product method [1], and its successor, iterative type analysis [8], use abstract interpretation for type analysis, where the program is executed, and the type signature of each encountered expression is generalized until it no longer violates the typing rules. We use this approach and extend it to handle multiple possible types of an expression and constant propagation in complex data structures such as lists and maps.

References

[1] O. Agesen. The Cartesian product algorithm. *Proc. of ECOOP*, 95:2–26, 1995.

[2] C. Allan, J. Tibble, P. Avgustinov, A. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, and G. Sittampalam. Adding trace matching with free variables to AspectJ. *Proc. of the 20th Annual ACM SIGPLAN Conf. on Object oriented Programming Systems Languages and Applications*, pages 345–364, 2005.

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

[4] J.-D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *POPL '93: Proc. of the 20th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 232–245, New York, NY, USA, 1993. ACM.

[5] H. Curry. *Combinatory Logic*. North-Holland, 1972.

[6] L. Damas and R. Milner. Principal type-schemes for functional programs. *Proc. of the 9th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 207–212, 1982.

[7] A. Deutsch. Interprocedural may-alias analysis for pointers: beyond k-limiting. *Proc. of the ACM SIGPLAN 1994 Conf. on Programming Language Design and Implementation*, pages 230–241, 1994.

[8] J. Eifrig, S. Smith, and V. Trifonov. Sound polymorphic type inference for objects. *Proc. of the tenth annual Conf. on Object-oriented Programming Systems, Languages, and Applications*, pages 169–184, 1995.

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

[10] S. F. Goldsmith, R. O'Callahan, and A. Aiken. Relational queries over program traces. In *OOPSLA '05: Proc. of the 20th Annual ACM SIGPLAN Conf. on Object Oriented Programming, Systems, Languages, and Applications*, pages 385–402, New York, NY, USA, 2005. ACM.

[11] M. Gorbovitski, T. Rothamel, Y. A. Liu, and S. D. Stoller. Efficient runtime invariant checking: A framework and case study. In *Proc. of the 6th Sixth Intl. Workshop on Dynamic Analysis*, Seattle, Washington, July 2008.

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

[13] J. Graver. *Type-Checking and Type-Inference for Object-Oriented Programming Languages*. PhD thesis, University of Illinois, 1989.

[14] M. Hind. Pointer analysis: haven't we solved this problem yet? In *PASTE '01: Proc. of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 54–61, NY, USA, 2001. ACM.

[15] R. Kazhankodathed. http://tinyurl.com/5b9qfe.

[16] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[17] W. Landi and B. G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Proc. of the Conf. on Programming Language Design and Implementation (PLDI)*, volume 27.7, pages 235–248, New York, NY, 1992. ACM Press.

[18] R. Lencevicius, U. Hölzle, and A. Singh. Dynamic Query-Based Debugging of OO Programs. *Automated Software Engineering*, 10(1):39–74, 2003.

[19] R. Lencevicius, U. Hölzle, and A. K. Singh. Dynamic query-based debugging. *Lecture Notes in Computer Science*, 1628:135–149, 1999.

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

[21] J. Lundberg and W. Lowe. A scalable flow-sensitive points-to analysis. *Compiler Construction–Advances and Applications, Festschrift on the occasion of the retirement of Prof. Dr. Dr. hc Gerhard Goos, Lecture Notes in Computer Science (LNCS), to appear in*, 2007.

[22] M. Martin, B. Livshits, and M. Lam. Finding application errors and security flaws using PQL: a program query language. *ACM SIGPLAN Notices*, 40(10):365–383, 2005.

[23] J. Plevyak and A. Chien. Precise concrete type inference for object-oriented languages. *Proc. of the Ninth Annual Conf. on Object-oriented Programming Systems, Language, and Applications*, pages 324–340, 1994.

[24] A. Potanin, J. Noble, and R. Biddle. Snapshot query-based debugging. *Proc. of Australian Software Engineering Conf.*, pages 251–259, 2004.

[25] G. Ramalingam. The undecidability of aliasing. *ACM Trans. Program. Lang. Syst.*, 16(5):1467–1471, 1994.

[26] D. Rémy and J. Vouillon. Objective ML: a simple object-oriented extension of ML. *Proc. of the 24th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 40–53, 1997.

[27] J. Robert and K. Viggers. Implementing Protocols Via Declarative Event Patterns. *ACM SIGSOFF Software Engineering Notes*, 29(6):1–21, 2004.

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

[29] J. Whaley and M. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. *Proc. of the ACM SIGPLAN 2004 Conf. on Programming language design and implementation*, pages 131–144, 2004.

[30] D. Willis, D. Pearce, and J. Noble. Efficient object querying for Java. *Proc. of the European Conf. on Object-Oriented Programming*, pages 28–49, 2006.

[31] D. Willis, D. Pearce, and J. Noble. Caching and Incrementalisation in the Java Query Language. Technical Report WPN07, 2007.