A System for Invariant-Driven Transformations

A Dissertation Presented

by

Michael Gorbovitski

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Doctor of Philosophy

in

Computer Science

Stony Brook University

May 2011

Stony Brook University

The Graduate School

Michael Gorbovitski

We, the dissertation committee for the above candidate for
the degree of Doctor of Philosophy
hereby recommend the acceptance of this dissertation

Yanhong A. Liu — Dissertation Advisor
Professor, Computer Science, Stony Brook University

Scott Stoller — Chairperson of Defense
Professor, Computer Science, Stony Brook University

Rob Johnson — Committee Member
Assistant Professor, Computer Science, Stony Brook University

John Field — External Committee Member
IBM, Manager of the Advanced Programming Tools Group

This dissertation is accepted by the Graduate School.

Lawrence Martin
Dean of the Graduate School

Abstract of the Dissertation

A System for Invariant-Driven Transformations

by

Michael Gorbovitski

Doctor of Philosophy

in

Computer Science

Stony Brook University

2011

Transformation systems are important for program manipulations such as optimization, instrumentation, and refactoring. Even though not always stated explicitly, these transformations are always driven by invariants, such as maintaining invariants for optimization, checking invariants for verification, and so on.

This dissertation describes a system that allows coordinated transformations driven by invariants to be specified declaratively, as invariant rules, and applied automatically. We specially describe our implementation for applying invariant rules to Python and C programs, alias and type analyses developed for applying invariant rules, and a method for composing and optimizing invariant rules. We also describe successful applications of the system in generating efficient implementations from clear and modular specifications, in instrumenting programs for runtime invariant checking, query-based debugging, and profiling, and in code refactoring.

Contents

Acknowledgments

First, I would like to thank my advisor, Prof. Annie Liu, for directing and guiding me in the pursuit of my doctorate of philosophy, for giving me ideas, helping with implementation, guiding and helping me with writing, and most importantly, for letting me not give up.

I would further like to thank Prof. Scott Stoller, Prof. Rob Johnson, and Dr. John Field for their interest in this work and for serving on my dissertation committee, and to separately thank Prof. R. Sekar for his feedback on this dissertation.

In addition, I would like to thank Prof. Stoller, a frequent collaborator on the work that composes this dissertation, for the clarity of thoughts he introduced during the process of publishing some results that this dissertation consists of, which greatly helped me in the formulation of this entire work; his comments on writing were a fundamental help in the presentation of this work.

I would like to greatly thank the members of the DAR lab for helping this work take its shape. Tom Rothamel, Tuncay Tekle, Katia Hristova, Yury Puzis, Eugene Borodin, Jon Brandvein, and Bo Lin — thank you! You helped, by collaborating with me outright on parts or all of this work, by offering support, and by giving useful advice.

Chapter 1

Introduction

1.1 Invariant-driven program transformations

Transformation systems are important for program manipulations such as optimization, instrumentation, and refactoring. Even though not always stated explicitly, these transformations are commonly driven by invariants, such as maintaining them for optimization, checking them for verification, and so on. Generally, we use the term invariant to refer to the equality of the results of two expressions that holds during program execution.

For example, for optimization, to quickly return the size of a collection of data, at all program points where elements are added or removed, we must add code that updates the variable that holds the size of the collection; the invariant is that the value of the variable equals the size of the collection. For another example, for instrumentation, to check that memory is managed correctly, at any program point where a reference is added to or removed from an object, we can insert code that checks whether the variable that holds the reference count of the object is incremented or decremented appropriately, and if not, prints an error message and stops the program; the invariant is that either the variable equals the number of references or the error message is printed and the program is stopped. For yet another example, for refactoring, for any code fragment that is the same as the body of a given method modulo a substitution for the parameters of the method, we can replace the code fragment with a call to the given method with arguments obtained from the substitution; the invariant is that each call to the method is equivalent to the corresponding replaced code fragment.

Maintaining or checking such invariants by hand is very tedious and error-prone due to the fact that the statements that update the values the invariant depends on are scattered throughout the program. This introduces two problems: (1) it increases the difficulty of maintaining the program by scattering the code that maintains or checks the invariant throughout the program, and

(2) it makes it easy to miss an update to some value the invariant depends on.

A solution to the first problem is to use a program transformation system so that the code that maintains or checks the invariant is written in one spot, and then distributed to the appropriate scattered updates in the program automatically. There is a vast amount of research on program transformation languages and a vast number of program transformation systems exist, as described in a number of surveys, e.g., [81, 105], and collected on the web [101]. Eminent systems include, e.g., APTS [80], KIDS [93], CIP [13], and Stratego [14].

Except for APTS, all these systems still suffer from the second problem: the transformation writer still has to painstakingly make sure that he does not miss a single type of update. APTS, unfortunately, only applies to a simple language with set expressions and statements in straight-line code.

We solve both of these problems by creating a program transformation system that detects all statements that potentially update a given invariant and makes sure that the program transformation being applied to maintain or check the invariant either handles all such updates, or is not applied at all — we call this performing an invariant-driven program transformation in a coordinated manner.

This dissertation describes:

- A program transformation language (InvTL) that allows for the declarative specification of coordinated transformations driven by invariants, called invariant rules. The language also allows explicit specification of cost considerations. It was initially developed by us in [73], further refined in [71], and almost reached its current state in [72].
- A program transformation system (InvTS) that automatically applies invariant rules to Python and C programs in a coordinated manner.
- The static analyses required for applying an invariant rule in a coordinated manner. This includes algorithms for control flow graph generation and type analysis of Python programs and flow- and trace-sensitive alias analysis for dynamic languages. These were initially described in [42, 43], and further developed in [41].
- Applications that use InvTS, including generating efficient implementations from clear and modular specifications [73, 71], runtime invariant checking [42], query-based debugging [43], instrumentation for detection of error-prone peers in BitTorrent, as well as code refactoring during the implementation of InvTS. We describe experiments showing the efficiency and effectiveness of InvTS for these applications.

- A method for composing invariant rules that makes the overall effects of the composed invariant rules clearer and easier to understand, allows optimizations of the composed rules, and allows overall faster application of composed invariant rules compared with applying smaller transformations rules separately. This method was developed in [40].

1.2 Invariant rules

An invariant rule declaratively specifies that an invariant holds if all updates to the values that the invariant depends on are certain kinds of updates, and the corresponding maintenance work is performed at each update. It can also specify additional conditions on the query and updates, and additional declarations needed for the maintenance.

For example, the rule in Figure 1.1 expresses that, to maintain the invariant that r equals the size of set s when every update that may affect the size of s is assigning s a new empty set, adding an element x to s, or removing an element x from s, the respective maintenance is assigning r the value 0, incrementing r by 1 if x is not in s before the addition, or decrementing r by 1 if x is in s before the removal; the cost of the original query is linear in the size of s, and the cost of each update and maintenance is asymptotically the same as the cost of evaluating x, denoted cost(x), assuming that the set operations used in the rule take constant time. Thus, the linear-time `size` query can be

```
inv $r = s$.size()              $O(|s|)$

 at $s$ = new set()             $O(1)$
 do $r$ = 0                     $O(1)$

 at $s$.add($x$)                $O(\text{cost}(x))$
 do before
    if not $s$.contains($x$):   $O(\text{cost}(x))$
      $r$ = $r$+1

 at $s$.del($x$)                $O(\text{cost}(x))$
 do before
    if $s$.contains($x$):       $O(\text{cost}(x))$
      $r$ = $r$-1
```

Figure 1.1: An invariant rule for set size.

replaced by a constant-time retrieval from r at no extra asymptotic cost in maintenance, regardless of the frequencies of queries and updates.

Expressing coordinated incremental maintenance of invariants using invariant rules is high-level and declarative, making the transformations easier to understand, use, extend, and verify. The semantics of the rules encapsulates many low-level, procedural details. For example, all updates to the parameters of a query must be detected, one way or another, even in the presence of object aliasing, and maintenance must be performed at all updates. This contrasts with traditional use of individual rewrite rules with programmed strategies for tree walking, program analysis, and rule applications.

Invariant rules can be put in a library and reused from application to application, as opposed to being re-discovered and manually embedded in scattered places in each application program. While it may be extremely difficult to manually maintain multiple scattered invariants under many scattered updates correctly, doing so by automatically applying a library of invariant rules is easy.

Core form of invariant rules. The core form of an invariant rule is:

$$\begin{array}{l}\textbf{inv}\ r = \textit{query}\\ (\textbf{at}\ \textit{update}\\ \ \textbf{do}\ \textit{maint})+\end{array} \tag{1.1}$$

where *query*, *update*, and *maint* are patterns for matching queries, updates, and maintenance operations, respectively. The "+" indicates that there may be one or more instances of the clause.

The semantics of an invariant rule is: if a query in a program matches the *query* pattern, and every update to the parameters of the query in the program matches at least one of the *update* patterns, then a fresh variable instantiating r is declared in the program, occurrences of the query are replaced with uses of that variable, and at every update to the parameters of the query, the maintenance corresponding to the matching *update* patterns is inserted. Note that if a rule does not handle some updates to the parameters of a query in a program, then the rule does not apply to the query and its updates. We say that a rule *preserves* the invariant $r = \textit{query}$, if (1) $r = \textit{query}$ holds after initialization of r and (2) for each pair of *update* and *maint*, if $r = \textit{query}$ holds, then it still holds immediately after execution of *update* and *maint*, for all instances of *query*, *update*, and *maint*. (We do not consider concurrency here.) It is easy to see that preserving an invariant is a property that can be checked individually for each rule.

In the core form above, the maintenance work corresponding to an update can be done either before or after the update; this is correct if the maintenance code does not use the values of the variables assigned to by the update. To accommodate maintenance code that uses the values of those variables, the `do`-clause may have the form:

$$\begin{aligned}&\textbf{do}\ \mathit{maint}?\\ &\quad(\textbf{before}\ \mathit{maint}_1)?\\ &\quad(\textbf{after}\ \mathit{maint}_2)?\end{aligned} \tag{1.2}$$

where *maint* can be done either before or after the update, maint_1 must be done before, and maint_2 must be done after. A "?" after a clause indicates that the clause may be omitted. We allow a `do`-clause to be omitted if no maintenance needs to be done at an update.

To facilitate cost consideration, an invariant rule may specify the costs of the query, updates, and maintenance, by including a cost annotation of the following form after each of them:

$$\textbf{cost}\ \ \mathit{cost} \tag{1.3}$$

We use asymptotic running time as the cost model, and we assume that standard hashing is used for set and map operations. Other cost models that consider running time with constant factors, space usage, etc. could also be used. For ease of reading, we omit the keyword **cost** and align the costs to the right.

For example, the invariant rule in Figure 1.1 has the core form.

Metavariables and metafunctions. Variables in the rules in italic font are *metavariables.*

An unbound metavariable in a query or update pattern may match any program syntax element such that its value matches all further occurrences of that metavariable in the pattern. How matching is performed is subject-language specific, and is described in detail in Chapter 2, Section 2.3.1.

For example, in the rule for set size in Figure 1.1, s and x are metavariables in the query and update patterns; s `= new set()` restricts s to match an lvalue, and s`.add(`x`)` restricts x to match an expression. Other parts of patterns that are displayed in teletype font match program text exactly.

The scope of a metavariable in the query pattern is the entire rule, called *query* scope. The scope of a metavariable that appears in an update pattern but not in the query pattern is the update clause and the corresponding maintenance clause, called *update* scope. When matching occurrences of a name in the program, the scoping rules of the program being transformed are followed. Our implementation of these scoping rules is described in Chapter 2, Section 2.3.1.

Metavariables not in the query and update patterns, including r, denote distinct names not used for other purposes in the program being transformed in the scopes of these names. Such a name can be introduced in any scope that contains all uses of the name in maintenance, but for program clarity and modularity, by default, it is introduced in the smallest of these scopes. In particular, if the query and all updates and maintenance are in the same method, then r is instantiated with a new local variable of that method; otherwise, r is instantiated with a new field of the class that contains the query.

Finally, functions may be used in rules to help specify application conditions and form new program text, as discussed in the following subsections. They are called *metafunctions* and are displayed in normal font.

For Python, we define, among others, the following metafunctions:

`class`(*expr*)	returns the enclosing class of *expr*
`type`(*expr*)	returns the type of *expr*
`isvar` (*expr*)	returns `true` iff *expr* is a variable
`cost` (*expr*)	returns an expression that evaluates to the cost of *expr*, if given $freq$ data
`update`(*expr*)	returns `true` if the current `at` clause updates *expr*
`alias`($var1, var2$)	returns `true` if $var1$ is aliased to $var2$

Conditions on query and updates. Conditions that must be satisfied by a query or an update matched by the `inv`-clause or an `at`-clause of a rule can be specified by an `if`-clause of the following form immediately after the `inv`-clause or `at`-clause, respectively:

$$\textbf{if}\, \textit{condition}+ \tag{1.4}$$

where *condition* is a boolean Python expression that can involve metavariables and metafunctions.

For example, a rule may maintain the size of a set only if elements of the set are of a certain type, or if the query is in a certain class. This condition involves only the matched query, and may be specified in an `if`-clause immediately after the `inv`-clause. For another example, a rule may maintain the size of a set only if all updates to the set appear in the same class as the query. This condition involves also matched updates, and may be specified in an `if`-clause immediately after each `at`-clause.

Conditions may use metavariables bound in the query and update patterns. For convenience, the special metavariable *query* refers to the matched query, and the special metavariable *update* under an `at`-clause refers to the matched update.

Conditions may also use metafunctions that provide syntactic and semantic information from program analysis; the metafunctions available are subject-language specific. In particular, metafunction alias(x, y), which returns whether x and y may alias each other, is used in detecting all updates that may affect a query result. It may also be used explicitly in conditions. It can be computed using the analysis in Chapter 4.

During rule application, the condition must evaluate to either `true`, `false`, or a boolean subject language expression that depends on metavariables and aliasing information about them, and that can be evaluated to `true` or `false` at runtime.

The `if` clause affects rule application differently depending on whether it is in *query* or *update* scope.

In *query* scope, i.e., guarding the `inv` clause, if the value of the `if` expression is either `false` or known only at runtime, the invariant rule is not applied. The rule is not applied because we maintain the value of a query in the face of all updates to the data it depends on, and if at some point at runtime the `if` condition evaluates to `false`, this means that the query should not be maintained at that point, even if the subject program updates the data the it depends on. This makes the maintained query result inconsistent. Thus, if we cannot determine at compile-time whether we maintain the query result or not, we assume we do not maintain it.

In *update* scope, i.e., guarding a particular `at` clause, if the value of the `if` expression is `true` or known only at runtime, we do apply the particular `de` and `do` clauses associated with the `at` clause, and surround the code inserted by the `do` clause with a guard that checks at runtime whether the `if` expression is actually `true`. In the cases where the value of the `if` metaexpression can be determined at compile-time to be `true`, the guard can be removed.

Declarations. An invariant rule may specify declarations needed for maintenance. Declarations used by maintenance under multiple `at`-clauses or a single `at`-clause may be specified by a `de`-clause of the following form after the `inv`-clause or the corresponding `at`-clause, respectively:

$$(\textbf{de}\ \ ((\textbf{in}\ \textit{scope}\ :)?\ \textit{declaration}+)+)* \tag{1.5}$$

where *scope* is a scope expression, defined below, in the invariant rule language that evaluates to a scope in the program being transformed, and each *declaration* is a declaration in the language of the program being transformed and may contain metavariables and metafunctions.

For example, a rule for maintaining set size may declare r to be a field in the class that contains the set size query in a `de`-clause after the `inv`-clause. For another example, a rule for maintaining the minimum of a set under element addition and deletion may declare a heap data structure in a `de`-clause after the `inv`-clause, and may use a `de`-clause after an `at`-clause to declare local variables used only within the maintenance code under that `at`-clause.

A scope expression has the form (*kind loc*)+, or **global**, where *kind* is **function**, **method**, **class**, **package**, or **file**, and *loc* is a Python expression augmented with metavariables and metafunctions; *loc* must evaluate at compile time to the name or unique identifier of a method, class, package, or file. For transforming programs in a given language, only the kinds allowed in that language may be used. An omitted kind uses as the default value the scope of the query or update pattern in the `inv`- or `at`- clause preceding the `de`-clause. For example, rules for transforming Python programs may use the scope expression

class `myset` **method** `add`

to indicate that local variables should be declared in method `add` of class `myset` of the default package. Specification of the scope for a list of declarations is optional. Recall that, by default, the smallest suitable scope is used.

If the variable, field, function, method, class, or package name in a `de`-clause is a metavariable not used in the query and update patterns, it denotes a distinct name not used for other purposes in the given program. Variables declared with global scope may be read and written from everywhere in the program; the implementation depends on the language of the program being transformed. Note that multiple maintenance clauses, and even multiple rules, may refer to the same declarations simply by using program text without metavariables.

In examples, we assume the language being transformed uses declarations of the form *name* : *type*. For example, to declare r of type `int` to be global, one may specify

de in global : r`: int`

and to declare r of type `int` in the class that contains the set size query, one may specify

de in class class(*query*) : r`: int`

where metafunction class(p) returns the enclosing class of the program syntax element p.

General form of invariant rules. In general, work can also be done at the query to help with incremental maintenance. Such work can be specified as a `do`-clause below the `inv`-clause. For example, to incrementally maintain the average of a set of numbers, one may incrementally maintain the sum and the count, and do a division right before the query, instead of doing the division immediately after the maintenance of sum and count.

We also allow the `inv`-clause to specify an equality between any two program syntax elements, not just a variable and a query expression. This is

$$\begin{array}{l}
\textbf{inv } \textit{result} = \textit{query} \\
(\textbf{if } \textit{condition}+)? \\
(\textbf{de } ((\textbf{in } \textit{scope} :)?\ \textit{declaration}+)+)* \\
(\textbf{do } \textit{maint}?\ (\textbf{before } \textit{maint})?\ (\textbf{after } \textit{maint})?)? \\
(\ \textbf{at } \textit{update} \\
\quad (\textbf{if } \textit{condition}+)? \\
\quad (\textbf{de } ((\textbf{in } \textit{scope} :)?\ \textit{declaration}+)+)* \\
\quad (\textbf{do } \textit{maint}?\ (\textbf{before } \textit{maint})?\ (\textbf{after } \textit{maint})? \\
\qquad\qquad (\textbf{instead } \textit{maint})?)?\ )+
\end{array} \tag{1.6}$$

Figure 1.2: General form of an invariant rule.

convenient, for example, if a query result is stored in a part of a data structure instead of a variable: the invariant may equate an expression that retrieves the query result with the query.

Finally, in the `do`-clause after an `at`-clause, the keyword **instead** can be used to indicate that an update matched by the update pattern should be replaced with the maintenance. This is useful when the update needs to be transformed. For example, the rule in Figure 1.1 has a problem: x can match any expression, not only a variable, and that expression will be evaluated both in the original call to `add` or `del` and in the maintenance; this is incorrect if x has side-effects. We can fix this problem, and reduce the maintenance cost to $O(1)$, by either adding a condition restricting x to match only variables, or replacing the `do`-clause under `add` with the following and changing the `do`-clause under `del` similarly:

```
do instead
  v = x
  if not s.contains(v):
    r = r+1
  s.add(v)
```

In summary, the general form of an invariant rule is given in Figure 1.2, where *query*, *result*, *update*, *declaration*, and *maint* are program text in the language of the program being transformed, except that they may contain metavariables and metafunctions; and *condition* and *scope* are a Boolean expression and a scope expression, respectively, in the invariant rule language. Cost may be specified for *query*, *result*, and each *update* and *maint*. We indicate metavariables with italic font, indicate metafunctions with normal font, and indicate program text with teletype font. When using pure teletype font, we indicate metavariables and metafunctions with a preceding "`$`", and indicate program text, possibly containing metavariables and metafunctions, with a pair of curly braces following a language indicator, for example `py{$s.size()}` for program text in Python containing the metavariable `$s`.

When to apply invariant rules. An invariant rule applies if (1) a computation in the program matches the *query* pattern, and the conditions after the `inv`-clause hold, (2) every update to the parameters of the query matches at least one *update* pattern, and the conditions after that `at`-clause hold, and (3) for optimization, the following cost condition holds for each matched update u, where $cost_q$ and $freq_q$, $cost_u$ and $freq_u$, and $mcost_u$ are the cost and frequency for the matched query q, the cost and frequency for update u, and the cost of the maintenance associated with update u, respectively:

$$\begin{array}{l} mcost_u \leq cost_u \text{ or} \\ \sum_{u \text{ where } mcost_u > cost_u} mcost_u \times freq_u \leq cost_q \times freq_q \end{array}$$

If frequency information is not available from analysis or profiling, the second disjunct can safely be ignored. If cost cannot be computed due to the elements of the cost expression being unknown, we assume the transformation will not increase the actual cost.

Examples of invariant rules. We give examples that show different usages of invariant rules and discuss developing and verifying invariant rules.

Incrementally maintaining join queries The rule in Figure 1.3 maintains the result of the query

```
{r: r in ROLES | (s,r) in SR, ((op,o),r) in PR}
```

under initialization and element addition and deletion for sets `ROLES`, `SR`, and `PR`. Given these sets and the values of `s`, `op`, and `o`, the query includes a role `r` from `ROLES` in the result set if the session-role pair `(s,r)` is in `SR`, and the permission-role pair `((op,o),r)`, where an operation-object pair is called a permission, is in `PR`. The query is used for the `CheckAccess(s,op,o)` operation in RBAC [3]. Its incremental maintenance was presented in pieces previously [71] without an expressive invariant rule language. `CheckAccess` is the most frequently used and most time critical operation in RBAC.

The incremental maintenance uses a map `MapSP2R` that maps any given values of `s`, `op`, and `o` to the desired set of roles. The inv-clause says to retrieve the query result from the map using `MapSP2R[(s,op,o])`, and it takes $O(1)$ time. Two additional maps are maintained: `SRMapR2S` is the inverse map of `SR`, and `PRMapR2P` is the inverse map of `PR`.

In the cost annotations, `SR21` denotes the maximum number of elements in the first component of `SR` for any element in the second component of `SR`, and similarly for `PR21`. Applying this rule allows the query to be done in minimum time, at the expense of more expensive updates.

```
inv MapSP2R[(s,op,o)] =                             $O(1)$
    {r: r in ROLES | (s,r) in SR, ((op,o),r) in PR}
                                                    $O(|\texttt{ROLES}|)$
 at ROLES = new set()                               $O(1)$
 do MapSP2R = new map()                             $O(1)$

 at SR = new set()                                  $O(1)$
 do MapSP2R = new map()                             $O(1)$
    SRMapR2S = new map()

 at PR = new set()                                  $O(1)$
 do MapSP2R = new map()                             $O(1)$
    PRMapR2P = new map()

 at ROLES.add(r)                                    $O(1)$
 do for s in SRMapR2S[r]:                           $O(\texttt{SR21*PR21})$
      for (op,o) in PRMapR2P[r]:
        if not MapSP2R[(s,op,o)].contains(r):
          MapSP2R[(s,op,o)].add(r)

 at SR.add((s,r))                                   $O(1)$
 do if ROLES.contains(r):                           $O(\texttt{PR21})$
      for (op,o) in PRMapR2P[r]:
        if not MapSP2R[(s,op,o)].contains(r):
          MapSP2R[(s,op,o)].add(r)
    SRMapR2S[r].add(s)

 at PR.add(((op,o),r))                              $O(1)$
 do if ROLES.contains(r):                           $O(\texttt{SR21})$
      for s in SRMapR2S[r]:
        if not MapSP2R[(s,op,o)].contains(r):
          MapSP2R[(s,op,o)].add(r)
    PRMapR2P[r].add((op,o))

 ...//deletion is the same as addition, except
    //without not in conditions and with add replaced by del
```

Figure 1.3: An invariant rule for a join query.

Profiling for frequency analysis. We describe how to automatically extend any invariant rule to generate instrumentation for profiling the frequencies of queries and updates, which helps justify incremental maintenance of the invariant. The extension has three steps: (1) under the `inv`-clause, declare a method `inccount`, from the package `invtslog`, that takes two parameters—the location of the query and `null` when a query is matched, and the locations of the corresponding query and the update when an update is matched—and counts the number of executions of each query and of each update for each query, (2) under the `inv`-clause, insert into the `do`-clause (creating the `do`-clause first if it does not exist) the call `invtslog.inccount(` loc(*query*)`,` `null)`, where metafunction loc(p) returns the unique location of the program syntax element p, and (3) under each `at`-clause, insert into the `do`-clause (creating the do-clause first if it does not exist) the call `invtslog.inccount(` loc(*query*)`,` loc(*update*)`)`.

For example, the invariant rule in Figure 1.1 is transformed into the rule in Figure 1.4.

```
inv r = s.size()                                       O(|s|)
 de in package invtslog:
    inccount(query,update):
      ... //increment count of query-update pair
 do invtslog.inccount(loc(query), null)

 at s = new set()                                      O(1)
 do ...//as before                                     O(1)
    invtslog.inccount(loc(query), loc(update))

 at s.add(x)                                           O(cost(x))
 do ...//as before                                     O(cost(x))
    invtslog.inccount(loc(query), loc(update))

 at s.del(x)                                           O(cost(x))
 do ...//as before                                     O(cost(x))
    invtslog.inccount(loc(query), loc(update))
```

Figure 1.4: An invariant rule for profiling set size and updates.

Refactoring. As a small example of refactoring, the invariant rule in Figure 1.5 renames a variable from `old` to `new` if the declaration of `old` is at a specified location, where metafunction decl(x) returns the program syntax element that declares x. The renaming respects scoping rules automatically. Conceptually, the rule matches all updates using *update* and does nothing at all of them, since no update affects the invariant. An efficient implementation simply omits matching updates.

```
inv new = old
 if loc(decl(old)) = ...//some specific location
 at update
```

Figure 1.5: An invariant rule for variable renaming.

The rest of the dissertation is structured as follows: Chapter 2 describes how InvTS is implemented and how it applies invariant rules to subject programs. Chapter 3 describes various applications of InvTS, including generating efficient implementations from clear and modular specifications [73, 71], runtime invariant checking [42], query-based debugging [43], as well as code refactoring during the implementation of InvTS. Chapter 4 describes the type and may-alias analyses required to find updates that effect a given invariant. Chapter 5 describes a systematic method for composing invariant rules, and for optimizing the composed rules. Each of these chapters includes descriptions of relevant related work, and, if relevant, experiments showing the effectiveness of the method described.

Chapter 2

System for applying invariant rules

InvTS is a program transformation system that takes a transformation specification, written as a set of invariant rules, and produces a transformed program according to the transformation specification.

This chapter presents the architecture of InvTS, the overall algorithm of how InvTS applies a set of invariant rules to the program to be transformed — the subject program, how InvTS performs incremental analysis to more efficiently apply a set of invariant rules, and experimental results. The chapter concludes by discussing related work on program transformation systems.

Applying a set of invariant rules includes parsing the invariant rules and programs, choosing which rules to apply when, and applying a single invariant rule with pattern matching and replacement.

2.1 System architecture and overall algorithm

InvTS architecture. InvTS is written in Python, and consists of (1) a core that provides subject-language independent services, including invariant rule parsing, the algorithm that transforms the subject program according to the parsed invariant rules, utilities, persistent storage, and (2) subject-language dependent language modules, called LMs, that provide pattern-matching and replacement, static analysis, subject language parsing, etc. The Python language module is fully implemented — the only unsupported clause is `cost` — and is written completely in Python. The C language module is experimental, is mostly written in Python, but uses two C/C++ components: (1) the `elsa` C++ [75] parser, and (2) the GCC plugin architecture developed by Callanan et al. [15], and an InvTS-specific GCC plugin, developed by Callanan and Gorbovitski. These are used for performing pattern matching and abstract syntax tree transformations of C programs.

The architecture of InvTS is shown in Figure 2.1.

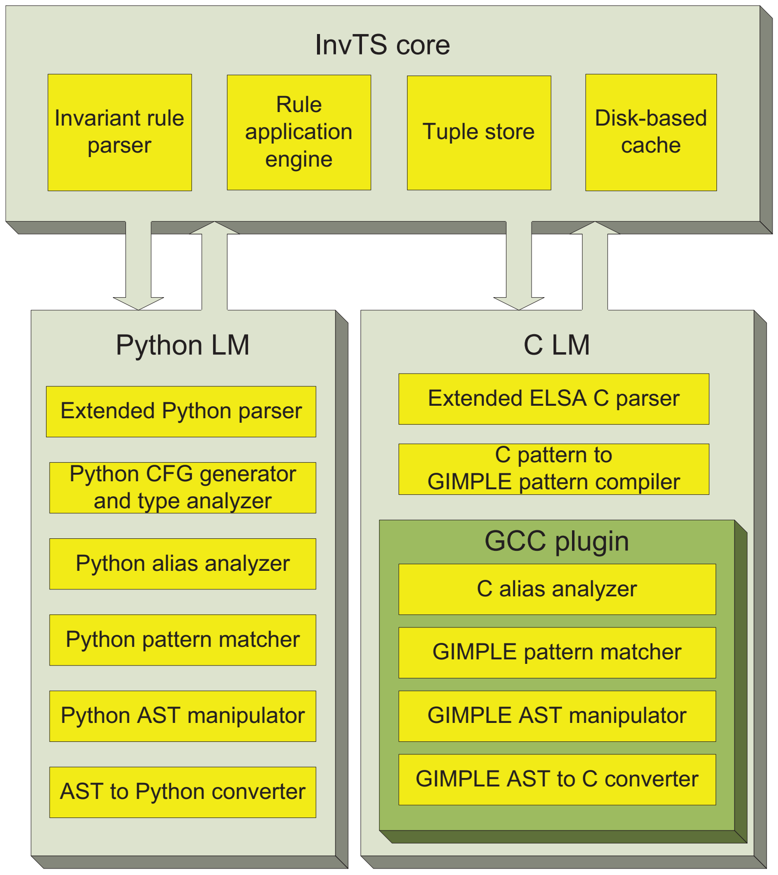


Figure 2.1: Architecture of InvTS.

The parsers, i.e., the invariant rule parser, the extended Python parser that supports metavariables and metafunctions, and the extended C parser that supports metavariables and metafunctions, are responsible for both loading and parsing invariant rules and Python and C programs. They are described in Section 2.2.

The rule application engine is responsible for applying a set of rules as described in the overall algorithm in Figure 2.2. The algorithm it uses to apply a single rule is described in Section 2.3.

The tuple store stores intermediate program analysis results in a manner that allows for incremental program analysis when applying a set of rules. It is described in Section 2.4.1.

The disk-based cache stores parsed rules, intermediate results of applying rules, and alias analysis results. It is described in Section 2.4.

The control-flow graph generation, type analysis, and alias analysis for

Python, and the alias analysis for C are responsible for the corresponding static analyses. They are described in Chapter 4.

The pattern matchers match query and update patterns, and are described in Section 2.3.2; how the AST is manipulated in transforming the subject program is described in Section 2.3.5.

---

1. Parse the given invariant rules and subject program:
   - (a) Parse the invariant rule file using the invariant rule parser, as described in Section 2.2, and store the parsed rules in *ruleset*.
   - (b) Parse the subject program file using the standard parser for the subject language, as described in Section 2.2, and store the parsed program in P.
2. Apply *ruleset* to P:
   - (a) Initialize *workset* to *ruleset*.
   - (b) If *workset* is not empty, pick any rule R from *workset*, and:
     - i. remove R from *workset*,
     - ii. clone P to P_{clone},
     - iii. apply R to P, as described in detail in Section 2.3; this succeeds if R's query pattern matches some query in P, all updates to the query match some of R's update patterns, and the condition specified in R's `if` clause is satisfied;
     - iv. if step iii succeeds, go to step 2(a); otherwise, replace P with P_{clone} and go to step 2(b).
3. Output P.

Figure 2.2: Overall rule application engine algorithm.

2.2 Parsing invariant rules and programs

Invariant rules are read in and parsed by the TPG parser generator [26] modified to handle patterns by using a custom lexer provided by the appropriate LM. Instead of first lexing the entire input and then parsing it, we lex only one token ahead of the parser. The custom lexer lexes the entire invariant rule file in a standard manner, except when it encounters a language indicator (`py` for Python, `C` for C) followed by an opening curly brace. In that case, starting from after the opening curly brace, it consumes the longest string that is a valid program string extended with metavariables and metafunctions in the language indicated by the language indicator, and ends on the closing curly brace. This consumed string, which is used as a pattern, is then returned to the parser.

After the invariant rule file is parsed using TPG, all returned patterns are parsed by a more sophisticated but slower parser that produces an internal representation. This representation is used for either matching patterns against the program to be transformed, or for instantiating patterns. In the Python LM, this is a modified `compiler` module from the Python standard library. In the C LM, this is a modified `elsa` C++ parser. In both cases, the modifications allow the parsing of the subject language extended with metavariables and metafunctions.

For efficiency, after the rules are parsed into the internal representation, they are reordered as follows: a rule r_1 is placed before a rule r_2 if applying r_1 can make r_2 applicable, i.e., the patterns of the `de` or `do` clauses in r_1 contain code that matches the query or update patterns in r_2, and not vice versa. Other than these heuristics, we preserve the order of the rules given to us.

Parsing programs. InvTS uses the built-in Python parser to load Python programs and convert them to abstract syntax trees (ASTs).

InvTS uses the built-in GCC parser to parse C programs and convert them to GIMPLE, an intermediate representation of C that GCC uses internally. InvTS uses a GCC plugin to export the GIMPLE representation from GCC to InvTS.

2.3 Applying a single rule

InvTS tries to apply, and applies, a single rule R to program P using the following algorithm:

1. Find an AST node in P that matches R's query pattern and has not matched any patterns in R before; if no match is found, then R is not applied. Pattern-matching is described in Section 2.3.2. As part of pattern-matching, bind the metavariables in the `inv` clause to the matched AST nodes, and bind the metavariable `$query` to the AST node of the matched query. Metavariable scoping and binding are described in Section 2.3.1.

2. Evaluate the condition in the `if` clause under the `inv` clause statically. It does not evaluate to `true`, R is not applied. Evaluating conditions is described in Section 2.3.3.

3. For each `de` clause under the `inv` clause, insert the clause's instantiated pattern into P in the scope specified in the `de` clause. If the pattern cannot be instantiated or the specified scope cannot be found, then R is not applied. Instantiating patterns is described in Section 2.3.5.

4. Insert the instantiated patterns in the `do before`, respectively `do after`, clauses under the `inv` clause into P immediately before, respectively after, the match from step 1. If the pattern cannot be instantiated, then R is not applied.

5. Using control-flow and may-alias analysis, determine all statements and expressions in P that may update the variables that are in the matched query from step 1. Section 2.3.4 describes how this is done.

6. For each update, u, determined in step 5, apply an `at` clause to u in the following manner:

   (a) Find an `at` clause, a, whose pattern is matched by u, and that InvTS has not yet tried to apply to u. If no such clause is found, then R is not applied. As part of the pattern-matching, bind all unbound metavariables that occur in a to the AST nodes they matched, and bind `$update` to the AST node of the update u.

   (b) Evaluate the condition in the `if` clause under the `at` clause statically, making use of control-flow and may-alias information. If the condition (including alias condition) evaluates to `true`, proceed to step 6(c). If the alias condition cannot be evaluated precisely at compile-time, insert code to check at runtime whether the alias condition holds, and then proceed to step 6(c). Otherwise, find another `at` clause to apply, i.e., go back to step 6(a).

   (c) For each `de` clause under the `at` clause, insert the `de`-clause's instantiated pattern into P in the specified scope. If the pattern cannot be instantiated or the specified scope cannot be found, find another `at` clause to apply, i.e., go back to step 6(a).

   (d) Insert the instantiated pattern in the `do before`, respectively `do after`, clause under the `at` clause into P immediately before, respectively after, u. For the `do instead` clause under the `at` clause, replace u with `do instead` clause's instantiated pattern. If the pattern cannot be instantiated or the instantiated pattern cannot be inserted into the subject program, find another `at` clause to apply, i.e., go back to step 6(a).

2.3.1 Metavariables

Metavariables augment concrete syntax to match, move around, and construct new AST nodes from existing AST nodes. In patterns, metavariables are indicated with a preceding `$` sign.

Scoping. The rules for scoping metavariables appearing in difference clauses are (for brevity, we refer to any of `do`, `do before`, `do instead`, and `do after` as `do`):

- The scope of a metavariable that first appears in the query pattern is the entire rule, called *query* scope.
- The scope of a metavariable that first appears in any `de` clause of the query is all `de` clauses of the query, and all following clauses of the rule.
- The scope of a metavariable that first appears in any `do` clause of the query is all `do` clauses of the query, and all update clauses of the rule.
- The scope of a metavariable that first appears in an update pattern is that update clause and the corresponding `if`, `de`, and `do` clauses, called *update* scope.
- The scope of a metavariable that first appears in any `de` clause of the update is all `de` and `do` clauses of that update.
- The scope of a metavariable that first appears in any `do` clause of the update is all `do` clauses of the update.

Binding. A metavariable becomes bound in one of three ways:

1. If it is in a query or update pattern, in an `inv` or `at` clause, that matches an AST node, then the metavariable is bound to the matched AST node.
2. If it is in a declaration or maintenance code pattern, in a `de` or `do` clause, then a fresh variable is created, and the metavariable is bound to the AST node for the variable. The fresh variable is created in the smallest scope possible, i.e., if the `do` clause is to add code inside a function, the fresh variable is local to that function; if the `de` clause is to add a field to a class or a variable to the global namespace, respectively, the fresh variable is a field in that class or a variable in the global namespace, respectively.
3. Metavariables `$query` and `$update` are special, and are bound to the AST nodes that match the query and update patterns, in enclosing `inv` and `at` clauses, respectively.

Once bound, a metavariable can be used in any clause in which the variable is in scope.

2.3.2 Pattern matching for Python and C

Python and C programs are matched against patterns in the `inv` and `at` clauses of invariant rules.

The InvTS core does not define how pattern matching should be implemented. It is the responsibility of the language module to implement pattern matching for a subject language.

Matching Python patterns

A pattern is a sequence consisting of four kinds of elements: (1) Python code, (2) bound metavariables, (3) unbound metavariables, and (4) metafunctions.

We restrict patterns used for matching Python code to patterns consisting of (1), (2), and (3) — in the current implementation, metafunctions are disallowed.

Literal Python code. Matching literal Python code is simple: convert the literal Python code to an abstract syntax tree (AST), and compare this AST with all subtrees of the subject program AST. We use the standard algorithm by Yang [111] to do AST matching in $O(n \times k)$ time where n and k are the sizes of the program AST and the pattern AST, respectively.

Unbound metavariables. What does it mean to match an unbound metavariable? Let us illustrate with an example: consider the pattern `f($v)` on the left of Figure 2.3, and the Python program fragment `f(x) g(y)` on the right.

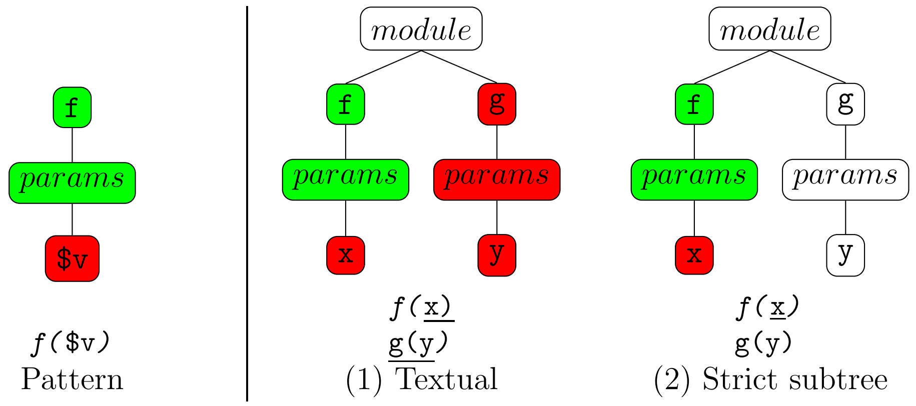


Figure 2.3: Textual vs. strict subtree pattern matching. Match is underlined.

It is clear that we want to match `f(x)`, binding `$v` to `x`. This rules out longest string matching, as it would bind `$v` to all the dark-grey parts of Figure 2.3(1), which is counter-intuitive. We may first consider that an unbound metavariable is bound to only a complete subtree of the AST — strict subtree matching. The results of strict subtree matching are shown in Figure 2.3(2): `$v` is bound to the parameters of `f`, which is what we wanted to accomplish.

Unfortunately, strict subtree matching has a drawback. Assume that we have the pattern `f(a,$v)` in Figure 2.4 on the left, and the Python expression `f(a,b,c)` in Figure 2.4 on the right. We would like to bind `$v` to `b,c`. As can be seen from Figure 2.4(1), there is no complete subtree that contains only `b,c`, and thus the pattern would not match anything. We rectify the problem

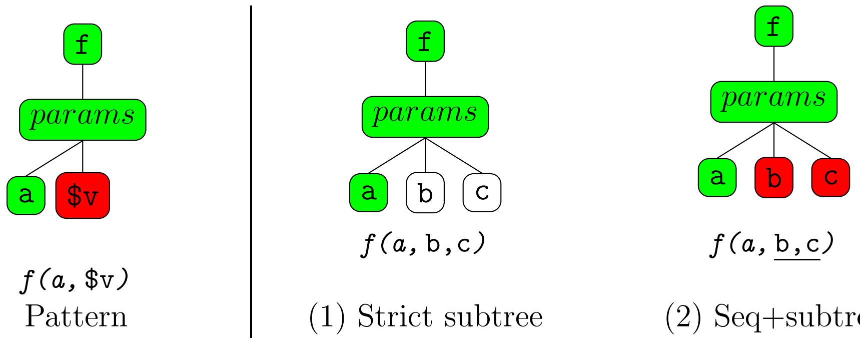


Figure 2.4: Strict vs. seq+subtree pattern matching. Match is underlined.

by allowing a metavariable to bind to either a complete subtree or a set of complete subtrees that is a consecutive sequence of children of an AST node. We call this seq+subtree matching. Figure 2.4(2) shows the result.

Unfortunately, even seq+subtree matching is not sufficient. Figure 2.5 shows it cannot bind `$v` to `b.c` in matching the pattern `a.$v` with expression `a.b.c`. To resolve this, we allow a metavariable to match not only a complete subtree, but also a complete subtree with up to one leaf node excluded and up to one node already matched by the non-metavariable part of the pattern. We call this op+seq+subtree matching. Figure 2.5(2) shows the result, with the light-shaded node (`a`) being excluded and the small dark-shaded node (`.`) matched by both `a.` and `$v`.

An unbound metavariable matched by a complete subtree becomes bound to the subtree. A metavariable matched to sequence of complete subtrees that are all children of a single AST node becomes bound to the sequence. A metavariable matched by a complete subtree with up to one leaf node excluded and up to one node already matched by the non-metavariable part of the pattern becomes bound to the complete subtree, with the excluded node tagged as *missing*, and the double-matched node tagged as *replacable*.

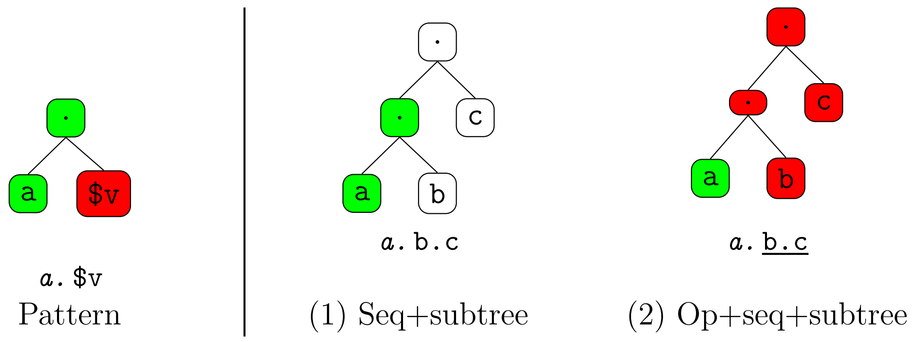


Figure 2.5: Seq+subtree vs. op+seq+subtree pattern matching. Match is underlined.

Bound metavariables. Pattern-matching a metavariable bound to a complete subtree with no nodes tagged is straightforward: replace the metavariable with the AST subtree it is bound to, reducing the match to a literal Python code match. Pattern-matching a metavariable bound to a subtree with a node tagged as *missing* or *replaceable* is done by treating that node as a phantom unbound metavariable that can only be matched using subtree matching.

Pattern-matching algorithm. We use a subtree-pattern-matching algorithm that supports wildcards, backreferences, and predicates [22]. It takes expected time $O(n^2 \times k)$, where n and k are the number of nodes in the program AST and pattern AST, respectively.

Matching C patterns

Literal C Code. Literal C code is matched by parsing the C code with ELSA, compiling it to GIMPLE, and using the GCC plugin [15] to perform subtree comparison on the GIMPLE representations of the pattern and of the subject program.

Unbound metavariables. For C, we only allow metavariables to match complete GIMPLE subtrees. For this task, the C LM uses the same subtree method as the Python LM: metavariables are bound to the complete GIMPLE subtree representing an expression or single statement in the subject program. This is performed by the GCC plugin as a tree expression match, with the unbound metavariable matched by any non-empty complete subtree.

Bound metavariables. Matching bound metavariables is done in the same way as matching literal C code.

2.3.3 Evaluation of conditions

The condition in an `if` clause is a metaexpression — an expression consisting of metavariables, metafunctions, and usual operators and functions expressed in the implementation language, Python. During rule application, a metaexpression must evaluate to `true`, `false`, or a Boolean expression in the subject language, to be evaluated to `true` or `false` at runtime.

The `if` clause affects rule application differently depending on whether it is in a *query* or *update* scope.

In *query* scope, i.e., under the `inv` clause, there are two cases:

- `true` — then the `inv` clause is applied.
- otherwise, the `inv` clause can not be applied. Thus, the subject program is not transformed by this rule application.

In *update* scope, i.e., under the `at` clause, there are three cases:

- `true` — then the `do` and `de` clauses are applied.
- `false` — then the `at` clause is not applied, and InvTS searches for another `at` clause to apply; if none is found, the subject program is not transformed by this rule application.
- known only at runtime, and depends solely on aliasing information — then InvTS inserts code to guard instantiated patterns of the `do` and `de` clauses so that they are only executed if the condition evaluates to `true` at runtime.

2.3.4 Detecting updates to values that the query depends on

Recall that `at` clauses are matched against *all* statements and expressions that can potentially update the parameters of the query specified. To apply a transformation rule, InvTS must detect *all* updates to the parameters of the query, even under object aliasing. It is impossible to do this completely at compile-time. Thus, when InvTS knows at compile-time that a statement *may* update a parameter of the query, it guards the corresponding maintenance code by a check that checks at runtime whether that statement updates that parameter of the query. As such runtime checks can be expensive, InvTS attempts to add as few of them as possible — it only introduces a runtime check if InvTS can statically determine that the statement *m*ay update the value of the parameter.

InvTS does the above in four steps:

1. For each parameter of each matched query, find all *syntactic updates*, i.e., syntactically determine all statements that may change that parameter.

2. For each syntactic update, derive the guard for checking at runtime whether the syntactic update actually changes the query parameter.

3. Prune syntactic updates, i.e., for each syntactic update, use static analysis to evaluate its derived guard to determine whether the syntactic update may change that parameter, and remove the update if it cannot.

4. Remove redundant guards, i.e., for each guard not removed in step 3, use static analysis to determine whether the guard must evaluate to `true`. If so, remove the guard.

Finding all syntactic updates. A syntactic update for a query parameter p is a statement for which, based solely on the statement's syntax, its type, and the matched query, InvTS can say that the statement may change p. Syntactic updates are solely determined by the containers that the matched query iterates over, and by the conditions in the matched query.

If p is a container over which the matched query iterates, and its type is t, then any statement that modifies a container of type t_1 s.t. t_1 is compatible with t is a syntactic update. The assignment to p in the matched query is also a syntactic update. If p is represented by `v.f`, where `v` is a variable and `f` is a field, then all assignments to `v` are syntactic updates, and all assignments to `v1.f`, where `v1` is any other variable whose type is compatible with the type of `v`, are syntactic updates. This is similarly extended to the case where `a` is not a variable, but is itself a field access, the case where p is represented by `v[i]`, and combinations thereof.

If p is a variable `v`, and `v` is in the condition of the matched query, then any statement that assigns to `v` is a static update. If the condition contains an expression `v.f`, where `v` is a variable and `f` is a field, then all assignments to `v` are syntactic updates, and all assignments to `v1.f` where `v1` is any other variable whose type is compatible with the type of `v`, are syntactic updates. As before, this is similarly extended to nested field accesses, indexing operations, and any combination thereof.

Finally, if p is an lvalue `lv`, i.e., a variable, a list element (`a[1]`), etc., then any statement that assigns to `lv` is a syntactic update.

Deriving guards for a syntactic update. Given a syntactic update u to a query parameter p, we derive the guard for u's maintenance code with respect to p as follows.

If u is a method call on an expression e, p is a variable `v`, and the type of `v` is compatible with the type of e, then, if e and `v` are the same object, and u

updates the value of that object, this may change the query result. Hence, the guard is `if v is` e; Similarly, if p is a field access `v.f`, where `v` is a variable and `f` is a field name, and the type of `v.f` is compatible with the type of e, the guard is `if v.f is` e. Such a guard is cheap, i.e., takes time O(1).

If u is an assignment to an lvalue `lv` and p is a field access `v.f`, where `v` is a variable and `f` is a field name, and the type of `lv` is compatible with the type of `v`, then the guard is `if v is lv`. Such a guard is cheap, i.e., takes time O(1).

If u is an assignment to an lvalue of the form `expr.f` where `expr` is an expression and `f` is a field name, and if p is either part of the `inv if` condition, or is iterated over in the query, and is a nested field access of the form `v.fs.f` where `v` is a variable, `fs` is a non-empty dot-separated sequence of field names, and `v` is an iteration variable in the query, iterated over the container `S`, then the guard is `for v in S: if v.fs is expr:`. This guard is equivalent to asking: does there exist an element in `S` that is aliased to `v.fs`? This guard has run-time complexity O($\#S$).

If u is an assignment to an lvalue of the form `expr[expr2]`, we derive for it the same guards as if u were an assignment to `expr.expr2`.

Pruning syntactic updates. A syntactic update may be pruned if: (1) the syntactic update's guard is never satisfied, or (2) the syntactic update can not affect the query result.

The guard of the syntactic update may be satisfied only if at least one `is` expression in it may evaluate to true at runtime. To verify this condition, we generate a new guard expression from the guard by replacing every `x is y` with `x may-alias y`, and replacing every `x in S` with `X intersect A !=` {} where `X` and `A` are the may-alias set of `x` and the union of the may-alias sets of all element in `S`, respectively. If, after static evaluation, the guard expression evaluates to `false`, the guard cannot be satisfied. Thus, this syntactic update is pruned from the set of updates that may change that parameter of the query.

If the guard expression did not evaluate to `false` at compile-time, then InvTS determines if the syntactic update can affect the query result. Given a syntactic update u to the parameter p, u can update the query result if the *query* that uses p is reachable from u following control flow. If that is not the case, the syntactic update is pruned from the set of updates that may change that parameter of the query.

Removing redundant guards. A guard is redundant if the static update u to the query parameter p is guaranteed to be an update to p, i.e., all `is` expressions in the guard evaluate to `true` at runtime.

To verify this, we generate a new guard expression from the guard by replacing `x is y` with `(x may-alias y and type(x)==type(y) and` `len(may-alias-set(x)) == 1 and len(may-alias-set(y)) == 1` and by replacing loops over a container with a conjunction over the union of the may-alias sets of the contents of the container. If the guard expression evaluates to `true` statically, the guard will always be satisfied at runtime, and thus is removed.

Given control-flow, type, and may-alias information for the subject program, it is easy to perform the above four steps. For C, there is a plethora of algorithms for obtaining control-flow, type, and may-alias information; for example, control-flow, type, and flow-insensitive may-alias information can be directly obtained from GCC. For Python, our static analysis algorithms for obtaining this information are presented in Chapter 4.

2.3.5 Instantiating patterns and inserting them into Python and C programs

How to insert declarations and maintenance code (i.e., apply the `de` and `do` clauses) are specified in each LM. For both Python and C, the overall steps for inserting declarations or maintenance code are the same:

1. instantiate the pattern that specifies the code to be inserted, and
2. insert the instantiated pattern into the program.

Instantiating Python patterns. The `de` and `do` clauses contain not only subject language code, but also metavariables and metafunctions. The patterns must first be instantiated, i.e., converted to Python code. We describe how bound and unbound metavariables are converted first:

- Unbound, i.e., the metavariable is unbound. (1) A fresh variable is created in the smallest scope possible. (2) The pattern AST node that represents the metavariable is replaced with the AST node representing the variable name.
- Bound - strict pattern, i.e., the metavariable is bound to a complete AST subtree. We replace the pattern AST node that represents the metavariable with the AST subtree that the metavariable is bound to.
- Bound - seq pattern, i.e., the metavariable is bound to a sequence of complete subtrees. If the metavariable in the pattern AST is bound to a node that, when replaced with a list of AST nodes, forms legal Python code, replace the pattern AST node with the bound sequence of AST nodes. Otherwise, instantiation fails.

- Bound - op pattern, i.e., the metavariable is bound to a complete subtree with up to one node tagged *missing*, and up to one node tagged *replaceable*, called M_{old} and R_{old}, respectively. We do the following steps: (1) based on the type of the node tagged as *missing* and the location of the metavariable in the pattern AST, determine which node of the program AST to use to replace M_{old}, denoted as M_{new}; (2) replace M_{old} with M_{new} in the pattern AST, resulting in a complete AST subtree; if there is a node tagged as *replaceable* in the pattern AST, perform the same, substituting R for M.

Metafunctions are functions defined in the LM that take zero or more ASTs as arguments and return an AST. As such, they are simply applied after metavariables are handled, and the return values of the metafunctions replace the call nodes in the pattern being instantiated.

Instantiating C patterns. Instantiating C patterns is similar to Python, but simpler. This is because bound metavariables can only be bound to complete subtrees, which are instantiated in the same way as Python metavariables bound to complete subtrees. Metafunctions are handled the same way as for instantiating Python patterns.

Inserting instantiated patterns. For `do`, `do before`, `do instead`, and `do after` clauses, the instantiated pattern is inserted respectively immediately after, immediately before, instead of, or immediately after the AST node that matched the pattern of the corresponding `inv` or `at` clause. In the case when this is impossible due to the matched pattern being an expression inside another expression, (1) the matched expression is factored out and assigned to a fresh local variable, and (2) the instantiated pattern is inserted immediately before or after the newly created assignment statement. If, due to a `do instead` clause, InvTS tries to replace an expression inside another expression with a statement, the corresponding `at` or `inv` application is aborted: for an `at` application, InvTS then searches for a different `at` clause to apply; for an `inv` application, it reverts the subject program to the original, and tries to find another `inv` rule to apply.

The `de` clause contains a scope specifier that is a metaexpression: an expression consisting of Python operators, metavariables, and metafunctions. When the metaexpression is evaluated by InvTS, it must return an AST node in the program AST. If it does so, the code specified in the `de` clause is inserted immediately before that node; otherwise, behaviour is the same as for a failed `do instead` clause.

2.4 Incremental analysis for applying a set of rules

Applying invariant rules involves applying rules repeatedly to the subject program until a fixed-point is reached. Before InvTS can apply any rule, it has to perform costly static analysis on the subject program. If the subject program and the rule being applied is not changed between different rule applications, we could save a lot of time by just reusing the results of previous applications of the rule to the program. We use a disk-based cache to store parsed rules, intermediate programs transformed by rule application, and the corresponding static analysis results.

Unfortunately, successfully applying an invariant rule most likely changes the subject program. The cache does not help when the program is changed by the rule application, or when the rules are changed. Thus, repeated costly static analysis would be performed on the subject program by successive rule applications. This section presents a method for reusing parts of the static analysis results for a program P when performing the same static analysis on a slightly modified version of P, called P'.

We consider analysis algorithms that (1) are flow sensitive, i.e., they store analysis results per CFG node, (2) can have their results represented as a per-node set, with the algorithm adding analysis results to these sets as it goes, and (3) are either monotonic, i.e., once an analysis added some analysis result to a CFG node, it will never remove it from that CFG node, or correct in the face of additions to the result set of previously removed elements from the result set. The alias analysis algorithm from Chapter 4 satisfies these conditions — it is monotonic. The type analysis algorithm without generalization is also monotonic; while it is non-monotonic when generalization is added to it, adding back removed types does not invalidate the correctness of the algorithm — thus it still satisfies condition (3).

The transformation of P to P' can be represented by a change to the CFG of P, G. The CFG of P', G', can be obtained by addition and removal of nodes and edges to or from G. Let R be the set of nodes pointed to by the edges removed from G in order to obtain G'. For analyzing P', we reuse:

1. from the analysis of P, all analysis results of nodes not reachable from any nodes in R, and
2. from the analysis of P, all analysis results added to any nodes in P by the analysis algorithm before it added any analysis results to nodes in R.

Determining R. Because InvTS transforms the program AST, it is easy to find out whether a given AST edge is *added* or *removed*, and tag it as such.

We modify the CFG generation algorithm in Chapter 4 so that whenever it emits a CFG node that is pointed to by a *removed* edge, the node is added to R.

Reusing the analysis results of unreachable nodes. For any node that is not reachable from R to the program, its analysis results remain trivially correct. We can thus reuse all the analysis results of all nodes that are not reachable from nodes in R. To do this, we modify the analysis algorithm to add only the set of nodes reachable from R to the worklist. The set of nodes reachable from R can be determined easily using a DFS in $O(N)$ time, where N is the size of G.

Reusing unaffected analysis results. We can reuse all analysis results added to any nodes in P by the analysis algorithm before it added any results to nodes in R. We find such results by (1) modifying the analysis algorithm so that it associates a timestamp with any analysis result it adds to a CFG node, (2) finding the earliest timestamp for any analysis result added to any node in R, called t_{min}, and (3) reusing all analysis results with timestamps less than t_{min}.

1. We modify the analysis algorithm to store a timestamp in addition to the other analysis data it stores. We do so as follows:

1. we set the timestamp to 0 before the analysis of P starts,
2. we increment the timestamp by 1 after each operation by the analysis that changes analysis results — we call the current timestamp *now*,
3. we do not reset the timestamp when the analysis of P' starts, and
4. when adding per-CFG-node analysis results, we store, for each analysis result, all timestamps at which it was added.

2. We find the earliest timestamp of any analysis result added to any node in R by iterating over all nodes in R, finding the smallest timestamp of any result added to each node, and taking the minimum of the smallest timestamps.

3. Given two analysis runs: A on P and A' on P', A and A' start and end at the timestamps $A^{start} = 0, A^{end}$ and $A'^{start} = A^{end} + 1, A'^{end}$, respectively. We replace the part of the analysis algorithm that checks whether a given result is in the per-node analysis results with the following:

1. for A, we check whether the result being tested was added to the per-node analysis results with a timestamp t such that $0 \leq t \leq now$, and

2. for A', we check whether the result being tested was added to the per-node analysis results with a timestamp t such that $0 \leq t < t_{min}$ or $A'^{start} \leq t \leq now$.

Generalizing reuse for k subsequent analyses. We generalize the above reuse to k transformations (P_1 transformed to P_2 transformed to ... transformed to P_k) and k subsequent analyses (A_1 to A_k) by:

1. computing the time ranges for reusing analysis results from the analyses $A_1 \ldots A_i$ of $P_1 \ldots P_i$ for analyzing A_{i+1} of P_{i+1} as follows: after the analysis A_i completes, we form L_i, a list of tuples that are valid time intervals to be reused for computing A_{i+1}, given L_{i-1}:

$$L_i = \begin{cases} L_{i-1} & \text{if } t_{min} \leq A_i^{start} \\ L_{i-1} :: (A_i^{start}, t_{min}) & \text{if } A_i^{start} \leq t_{min} \leq A_i^{end} \end{cases}$$

   where t_{min} is computed as follows:

```
$t_{min}$=$now$
for (start,end) in $L_1$ .. $L_{i-1}$:
  for n in $R$:
    t = timeoffirstadditionion(n,(start,end))
    $t_{min}$=min($t_{min}$,t)
```

   where `timeoffirstadditionion(n,(start,end))` is a function that returns the timestamp of the first addition of any analysis result for the node `n` during the time interval `(start,end)`. The data structure used to implement the algorithm efficiently is described in Section 2.4.1, and

2. for A_j, we determine whether the analysis result r is valid for node n by checking whether there exists an addition of r to n with a timestamp t such that either $A_j^{start} \leq t \leq now$, or t is in one of the intervals in L_{j-1}.

2.4.1 Data structure

The static analysis algorithms we use all store data in pairs `(n,r)`, where `n` is an AST or CFG node identifier, and `r` is some arbitrary data. To implement the algorithm described above, we use a data structure that mimics a set of pairs, and supports the following operations in an efficient manner:

The rest of this subsection describes the data structure, its space complexity, how the operations are implemented, and their time complexity.

Each AST or CFG node n contains a map that maps the analysis result r added to n to an ordered list of timestamps at which r was added to n. The ordered list is represented as an array. We use the notation n`[`r`]` to denote accessing the array for a particular n and r, an O(1) operation.

Operation	*Description*
`add((n,r))`	add the pair `(n,r)` at *now*
`set((start,end))`	the set of pairs that have been added into the set between the times `start` and `end`
`contains(pair,(start,end))`	returns whether `pair` was added during the time interval `(start,end)`
`timeoffirstaddition(n,(start,end))`	the timestamp of the first addition of a pair with the first element bound to `n` during the time interval `(start,end)`

Figure 2.6: Required operations.

Addition. Adding the analysis result r to node n happens only at time *now*. As *now* is monotonically increasing, we know that r has never been added to n with a timestamp greater than *now*. Thus, we append *now* to the n`[`r`]` array.

It is clear that the time complexity of this operation is O(1), and the size of the n`[`r`]` array is a — the total number of additions of r to n, i.e., at most the number of steps taken by the algorithm.

Membership testing.

`contains`. To verify that the analysis result r has been added to node n in a given time range, we perform a standard binary search on the n`[`r`]` array. The time-complexity of `contains` is $O(log(a))$.

Aggregate operations.

`set`. Given a time range, generating the set of valid (n, r) tuples in that range requires going over all analysis results, and for each result, performing a `contains` call over the relevant range. The time complexity is $O(t \times (log(a)))$, where t is the number of distinct analysis results inserted in that time range.

`timeoffirstaddition(n,range)`. This returns the smallest timestamp of any analysis result added to `n` in `range`. To efficiently evaluate this function, in addition to maintaining a per-node map from each analysis result to the array of timestamps, we also maintain a per-node array of timestamps of any additions to the node. Thus, this function takes $O(log(a))$ time, adding an O(1) factor to all additions.

	Alias analysis time (s)	Type and CFG analysis time (s)	Total time (s)
Non-incremental analysis	36	9	48
Incremental analysis	14	7	24

Table 2.1: Analysis and transformation time for applying 21 rules with 114 `at` clauses to Constrained RBAC.

2.4.2 Experiments

To show that using incremental analyses reduces the time taken by InvTS to transform subject programs, we apply a set of invariant rules to an executable specification of Constrained RBAC using incremental and non-incremental analysis.

The executable specification of Constrained RBAC is written in 381 lines of Python. Incrementalization is performed by 21 invariant rules containing 114 `at` clauses total, written in 1137 lines of InvTL. After incrementalization, the transformed program is 2183 lines of Python. Table 2.1 shows that incremental analysis reduces the time taken by InvTS to do the transformation. The measurements were taken when running the analysis using Python 2.6.4 under Windows 7 on a Core i7 860 2.8GHz with 12 GB of memory, of which around 8GB was free when running our programs.

2.5 Related work

There has been a great amount of research and development done on program transformation systems [101, 39, 104, 24], but to our knowledge, there were no previous transformation systems that support all of the features that InvTL/InvTS supports: (1) coordination — the ability to apply a set of transformations together or not at all, (2) identification of all program fragments that may affect an invariant to make sure that the invariant is preserved, and (3) pattern matching and concrete syntax — a must for ease of writing and maintainability.

Coordination. Current state-of-the-art program transformation systems like Stratego/XT [104] — a language and toolset for program transformation that supports user-specified rewriting strategies that grew out of the need to add custom rewriting strategies to ASF+SDF [29, 102], TXL [23] — a transformation system that supports rule application to arbitrary ASTs in a fully automated manner, and Rhodium [68] — a source-to-source transformation system for writing compiler optimization, include a dazzling array of capabilities whose main purpose is to facilitate the easy writing of correct rules by the programmer, but they all lack coordination [39]. For example, while both Stratego/XT and TXL support multiple rules and support applying rules at once, the only way to implement a *coordinated* transformation in them is to manually write a strategy to do so. This is error-prone, as the rule writer may, for example, miss an update, and thus perform only part of the transformation.

Pattern matching and concrete syntax. Support for pattern matching and concrete syntax is widespread in program transformation languages. It is supported in systems such as TXL, Stratego, and ASF+SDF, but not supported in AspectJ or Rhodium. While InvTL does not make contributions in this arena, its syntax deviates as little as possible from the subject language, allowing the programmer to leverage his knowledge of the subject language.

Chapter 3

Applications

We used InvTL and InvTS for a wide range of applications. Figure 3.1 summarizes 24 examples grouped by whether the purpose is optimization, runtime verification, debugging, other instrumentation, refactoring, or other transformations.

This rest of this chapter presents a representative subset from each group of examples: Section 3.1 describes how we perform optimization by using InvTS to generate efficient implementations from clear and modular specifications of Core RBAC and various other smaller specifications [73, 71]; For runtime verification, Section 3.2 describes how we use InvTS to perform runtime invariant checking [42]; Section 3.3 describes how we use InvTS for query-based debugging [43]; Section 3.4 describes the use of InvTS for instrumentation and other transformations [73, 72]. Finally, Section 3.5 discusses related work.

3.1 Generating optimized implementations

What programs do on data can be classified as, or decomposed into, two kinds of operations: queries and updates, where queries compute results using data, and updates change data. For a simple example, consider the `LinkedList` class in Java 1.5. It has a query method `size` that returns the number of elements in the list, 12 other query methods that return elements, element indices, etc., and 15 update methods that add or remove elements.

How to implement the queries and updates can vary dramatically. In a straightforward implementation, each method performs its respective query or update. In the `LinkedList` example, `size` can iterate over the list, and each update method can simply do its addition or removal. This is clear and modular, but can have poor performance when such queries are performed frequently. A sophisticated implementation can maintain the results of these queries—i.e., maintain the invariants that the values retrieved from certain variables equal the results of these queries—incrementally with respect to up-

Use	Application	Program	Lang
O	Core RBAC	Core RBAC spec	py
	Constrained RBAC	Constrained RBAC spec	py
	Graph Reachability	test program	py
	Join Query	test program	py
	Wireless Protocol	test program	py
	Set Size Demo	test program	py
V	SMB Valid Ticket	pysmb	py
	SMB Repeated Auth	pysmb	py
	BitTorrent Peer No Dup	BitTorrent Peer	py
	BitTorrent Peer No Mod	BitTorrent Peer	py
	BitTorrent No Mismatch	BitTorrent Mainline	py
	InvTS No Shared Child	InvTS	py
	InvTS Own Child	InvTS	py
D	DOM Valid Parent	lxml benchmarks	py
	DOM No Shared Child	lxml benchmarks	py
	DOM Exception Cause	lxml benchmarks	py
	FTP Client	nftp	py
I	File Access Profiling	test program	py
	Reference Counting	test program	py
	Memory Coverage	ViM 7.0	c
R	InvTS Refactoring	file Rule.py in InvTS	py
	Variable Renaming	ViM 7.0	c
T	InvTS/py Test Suite	test program suite	py
	InvTS/c Test Suite	test program suite	c

O: optimization. V: runtime verification. D: debugging.
I: instrumentation. R: refactoring. T: other transformation.

Figure 3.1: Example applications.

dates to the query parameters—i.e., variables or fields on which the queries depend. In the `LinkedList` example, the result of `size` may be maintained in a field and simply be returned when queried. This is efficient, but no longer clear and modular, because each of the 15 update methods must also update this field appropriately.

This conflict between clarity and efficiency is much worse for complex systems with many queries and updates, where queries may involve objects from different classes, and updates may be spread in many classes. A query can be affected by many updates, and an update can affect many queries. It poses a serious challenge to consider all the complex dependencies and trade-offs and decide where and how to maintain what invariants. The resulting code can be significantly more difficult to understand.

We resolve this conflict by automatically transforming straightforward yet inefficient implementations into efficient yet sophisticated implementations. We do this by expressing these transformations as invariant rules (written in InvTL), and using InvTS to automatically apply them in a coordinated manner.

3.1.1 Core RBAC

Role-based access control (RBAC) is a framework for controlling user access to resources based on roles. It can significantly reduce the cost of security policy administration, is an ANSI standard [3], and is increasingly widely used in large organizations. Despite much research on RBAC, it is nontrivial to develop efficient implementations, and it is even harder to develop efficient implementations with precise complexity guarantees [71].

Core RBAC contains the following sets and relations, explained below, and the operations on them summarized in Figure 3.2.

```
  OBJS:     set(Object)    // an operation-object pair
  OPS:      set(Operation) // is called a permission.
  USERS:    set(User)
  ROLES:    set(Role)
  PR:       set(tuple(tuple(Operation,Object),Role))
  UR:       set(tuple(User,Role))
            // PR subset (OBJS * OPS) * ROLES
            // UR subset USERS * ROLES
  SESSIONS: set(Session)
  SU:       set(tuple(Session,User))
  SR:       set(tuple(Session,Role))
            // SU subset SESSIONS * USERS
            // SR subset SESSIONS * ROLES
```

A system has sets of objects, operations, users, roles, and sessions; their elements are of types `Object`, `Operation`, `User`, `Role`, and `Session`, respectively. A operation-object pair, called a permission, denotes an allowed operation on an object. A permission-role pair in `PR` denotes a permission assigned to a role. A user-role pair in `UR` denotes a user assigned to a role. A session-user pair in `SU` denotes a session and the unique user of the session. A session-role pair in `SR` denotes a session and a role active in the session.

Administrative commands. The following operations each adds an element to a set or a relation.

```
AddUser(user):
  pre-cond: user notin USERS;
  USERS = USERS + {user}
```

administrative commands	add/delete user/role, assign/deassign user, grant/revoke permission
supporting system functions	create/delete session, add/drop active role, check access
review functions	assigned users/roles
advanced review functions	role/user permissions, session roles/perms, role/user ops on obj

Figure 3.2: Functionalities of Core RBAC by categories.

```
  USERS add= user

AddRole(role):
  pre-cond: role notin ROLES;
  ROLES = ROLES + {role}

AssignUser(user,role):
  pre-cond: user in USERS, role in ROLES, [user,role] notin UR;
  UR = UR + {[user, role]}

GrantPermission(operation, object, role):
  pre-cond: operation in OPS, object in OBJS, role in ROLES,
            [[operation,object],role] notin PR;
  PR = PR + {[[operation,object],role]}
```

Deleting an element is symmetric to adding an element, but possibly with two kinds of additional updates. First, if an element is deleted from a set, then from all relations defined using the set, all tuples that contain the deleted element must be deleted. Second, `DeleteUser`, `DeleteRole`, and `DeassignUser` also delete the associated sessions, to satisfy the constraint that a session can have a role active only if the user of the session is assigned that role.

```
DeleteUser(user):
  pre-cond: user in USERS;
  UR = UR - {[user,r]: r in ROLES}
  for s in SESSIONS | [s,user] in SU:
    DeleteSession(user,s) // DeleteSession defined below
  USERS = USERS - {user}

DeleteRole(role):
  pre-cond: role in ROLES;
  PR = PR - {[[op,o],role]: op in OPS, o in OBJS}
  UR = UR - {[u,role]: u in USERS}
  for s in SESSIONS, u in USERS | [s,u] in SU, [s,role] in SR:
    DeleteSession(u,s)
  ROLES = ROLES - {role}

DeassignUser(user, role):
  pre-cond: user in USERS,role in ROLES,[user,role] in UR;
  for s in SESSIONS | [s,user] in SU, [s,role] in SR:
    DeleteSession(user,s)
  UR = UR - {[user,role]}

RevokePermission(operation, object, role):
  pre-cond: operation in OPS, object in OBJS, role in ROLES,
            [[operation,object],role] in PR;
  PR = PR - {[[operation,object],role]}
```

Supporting system functions. `CreateSession` creates a session for a user with an initial set of active roles; it first checks that the user is assigned those roles, and then adds the appropriate elements to `SU`, `SR`, and `SESSIONS`. `DeleteSession` deletes all elements of `SU`, `SR`, and `SESSIONS` that are associated with the session.

```
CreateSession(user, session, ars):
  pre-cond: user in USERS, session notin SESSIONS,
            ars subset AssignedRoles(user);
            // AssignedRoles is defined below
  SU = SU + {[session,user]}
  SR = SR + {[session,r]: r in ars}
  SESSIONS = SESSIONS + {session}

DeleteSession(user, session):
  pre-cond: user in USERS, session in SESSIONS, [session,user] in SU;
  SU = SU - {[session,user]}
  SR = SR - {[session,r]: r in ROLES} // maintain SR
  SESSIONS = SESSIONS - {session}
```

Adding and deleting active roles adds to and deletes from `SR` , respectively; adding an active role also first checks that the user of the session is assigned that role.

```
AddActiveRole(user, session, role):
  pre-cond: user in USERS, session in SESSIONS, role in ROLES, [session,user]
            in SU, [session,role] notin SR, role in AssignedRoles(user);
  SR = SR + {[session,role]}

DropActiveRole(user, session, role):
  pre-cond: user in USERS, session in SESSIONS, role in ROLES, [session,user]
            in SU, [session,role] in SR;
  SR = SR - {[session,role]}
```

`CheckAccess` checks whether an operation on an object is allowed in a session, i.e., whether there is a role that is active in the session and is assigned the operation-object pair as a permission.

```
CheckAccess(session, operation, object):
  pre-cond: session in SESSIONS, operation in OPS, object in OBJS;
  return {r in ROLES | [session,r] in SR, [[operation,object],r] in PR} != {}
```

Review functions and advanced review functions. These functions are queries on the basic sets and relations. Some (`AssignedUsers`, `AssignedRoles`, `RolePermissions`, `SessionRoles`) are over one relation, i.e., given a value for the left or right component of a relation, find all associated values for the other component in the relation. For example, the first two are review functions defined by:

```
AssignedUsers(role):
  pre-cond: role in ROLES;
  return {u: u in USERS | [u,role] in UR}

AssignedRoles(user):
  pre-cond: user in USERS;
  return {r: r in ROLES | [user,r] in UR}
```

Two functions (`UserPermissions`, `SessionPermissions`) are over two relations, i.e., given a value for one component of a relation, equate the other component of the relation with one component of a second relation, and find all associated values for the other component of the second relation. Two other advanced review functions (`RoleOperationsOnObject`, `UserOperationsOnObject`) require nested tuples but are otherwise similar to the above functions. All advanced review functions are defined below:

```
RolePermissions(role):
  pre-cond: role in ROLES;
  return {[op,o]: op in OPS, o in OBJS | [[op,o],role] in PR}

UserPermissions(user):
  pre-cond: user in USERS;
  return {[op,o]: r in ROLES, op in OPS, o in OBJS |
          [user,r] in UR, [[op,o],r] in PR}

SessionRoles(session):
  pre-cond: session in SESSIONS;
  return {r: r in ROLES | [session,r] in SR}

SessionPermissions(session):
  pre-cond: session in SESSIONS;
  return {[op,o]: r in ROLES, op in OPS, o in OBJS |
        [session,r] in SR, [[op,o],r] in PR}

RoleOperationsOnObject(role, object):
  pre-cond: role in ROLES, object in OBJS;
  return {op: op in OPS | [[op,object],role] in PR}

UserOperationsOnObject(user, object):
  pre-cond: user in USERS, object in OBJS;
  return {op: r in ROLES, op in OPS | [user,r] in UR, [[op,object],r] in PR}
```

3.1.2 Incrementalizing Core RBAC

Straightforward implementations of many operations in Core RBAC are inefficient because they involve iterating through sets from scratch. Efficient implementations require that the results of such expensive computations be stored, retrieved quickly when needed, and maintained incrementally when the sets that these results depend on are updated.

Identifying and incrementalizing expensive computations. We consider all operations that are not constant time expensive. These include set comprehensions, loops over sets, a subset test, a set union, and set differences in the Core RBAC specification. Once subset tests, set unions, and set difference operations are converted to loops over sets, the only remaining expensive computations are set comprehensions. Figure 3.3 lists all 16 occurrences of them, where the first column is the containing method, and last column classifies them into 9 different kinds of queries—`1x` for queries over one relation, and `2x` for queries over two relations.

We derive the incrementalization rules for each query for each kind of update to a parameter of the query using the method from [71]. The method accounts for two kinds of updates: updates to sets that the query depends on (based parameters), and updates to parameters of the query that can be set to any value. Maintenance code that incrementally maintains the result set when based parameters change is generated; for non-based parameters, a map that maps the values of those parameters to the results of the query is maintained. Then, the generic code for maintaining the result set by iterating over both the sets enumerated and the sets tested for membership in the subquery is generated. The specialized maintenance clause for handling additions and deletions to parameters of the query from the general maintenance clause is generated via four steps: (1) eliminate the loop over the set that is being added or deleted an element, because only the element being added or deleted needs to be considered for this loop in the incremental maintenance, (2) replace each loop whose loop variables are all bound with a test on the loop variables, because bound variables are filters of the values, (3) use auxiliary maps in

containing method	expensive query	kind
DeleteUser	{r: r in ROLES | [user,r] in UR}	1a
	{s: s in SESSIONS | [s,user] in SU}	1b
DeleteRole	{[op,o]: op in OPS, o in OBJS | [[op,o],role] in PR}	1c
	{u: u in USERS | [u,role] in UR}	1b
	{[s,u]: s in SESSIONS, u in USERS | [s,u] in SU, [s,role] in SR}	2e
DeassignUser	{s: s in SESSIONS | [s,user] in SU, [s,role] in SR}	2d
DeleteSession	{r: r in ROLES | [session,r] in SR}	1a
CheckAccess	{r: r in ROLES | [session,r] in SR, [[operation,object],r] in PR}	2c
AssignedUsers	{u: u in USERS | [u,role] in UR}	1b
AssignedRoles	{r: r in ROLES | [user,r] in UR}	1a
RolePermissions	{[op,o]: op in OPS, o in OBJS | [[op,o],role] in PR}	1c
UserPermissions	{[op,o]: r in ROLES, op in OPS, o in OBJS | [user,r] in UR, [[op,o],r] in PR}	2a
SessionRoles	{r: r in ROLES | [session,r] in SR}	1a
SessionPermissions	{[op,o]: r in ROLES,op in OPS,o in OBJS | [session,r] in SR,[[op,o],r] in PR}	2a
RoleOperationsOnObject	{op: op in OPS | [[op,object],role] in PR}	1d
UserOperationsOnObject	{op: r in ROLES, op in OPS | [user,r] in UR, [[op,object],r] in PR}	2b

Figure 3.3: Expensive queries in Core RBAC.

loops that have both bound and unbound loop variables to iterate over only the values of the unbound variables, (4) update an auxiliary map when its corresponding set of tuples is updated.

The generated rule for maintaining the `CheckAccess` is given in Chapter 1, Figure 1.3. As the rule is quite long, we do not reproduce it here.

3.1.3 Experiments

Core RBAC. To help confirm the correctness of the transformations and the complexity analysis results presented above, we first developed a straightforward implementation of Core RBAC that precisely follows the specification; we then applied our transformational method to it, both manually and automatically, using a number of different combinations of incrementalization rules, and we performed many experiments on the resulting implementations. All experimental results confirmed our expectations.

We can get a sense of how much effort using InvTS saved us by comparing the size of the straightforward program to the size of the incrementalized ones. We report the number of interesting lines of code, defined as non-empty and non-comment lines. The straightforward program consists of 125 lines of interesting code, including 16 expensive queries that could be incrementally maintained. When all 16 queries are incrementalized, the code more than tripled in size to 486 lines.

We developed a program that lets us perform black-box testing on both an original and an incrementalized implementations, to confirm that they produce the same output. It generates a sequence of random RBAC operations that are applied to both implementations. When an operation produces a result, the results produced by both implementations are compared and verified to be identical. This lets us automatically test the incrementalized program against the original program it was generated from. All tested implementations produced identical results for a sequence of 50 million operations, giving us confidence in the correctness of the incrementalized implementations.

We developed another program to generate data in a way that is governed by one or more independent variables. We used this program to generate a number of sets of input data, varying in some parameter, for each of which we need to determine the running time of each program. For each particular input and program, we compute the running time by running the program repeatedly on the data until the standard deviation of the set of running times is less than 10 percent of the mean of the set of running times. In all cases, the test programs were run a minimum of 10 times.

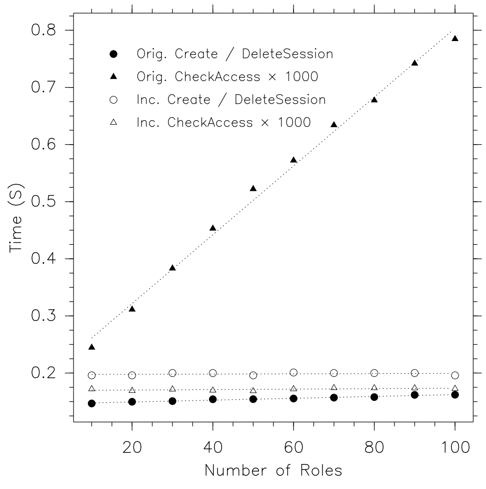
Our test programs are single-threaded, and were run underWindows XP SP2 on a dual-processor Athlon XP 2.8Ghz with 2 GB of memory, of which around 750 MB was free when running our programs. All of the experiments, written in Python, were run under ActivePython 2.4 Build 244. This system was also used to run the incrementalizer, which took around 30 seconds to complete the incrementalization of RBAC.

We compare the performance of the straightforward implementation and the incrementalized implementations. We measure the time it takes each implementation to complete 1000 repeats of a simple operation pattern. This pattern consists of the creation of a session with 10 active roles, 1000 random access checks, and the deletion of the session. The complexity analysis predicts that the operations and parameters that dominate the asymptotic running time differ between the straightforward and incrementalized implementations. In the straightforward implementation, the asymptotic time of all functions should be linear in the number of roles, as access checks and session creation and deletion are all linear in the number of roles. In the incrementalized implementation, `CheckAccess` should be constant time, and the asymptotic time should be dominated by the cost of `CreateSession` and `DeleteSession`, which is linear in the number of permissions assigned to the roles activated in a session.

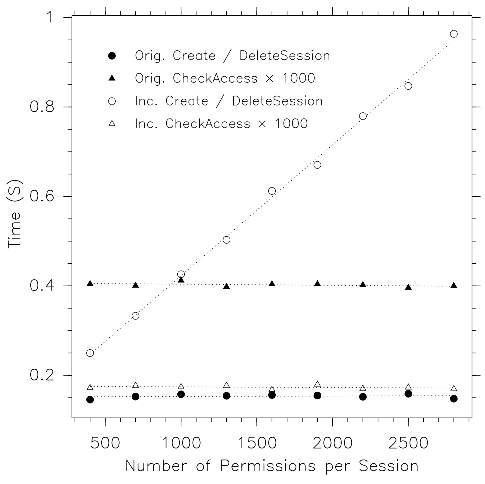
Figure 3.4(a) compares the performance of the implementations where the number of permissions per session is fixed at 100 and number of roles varies. It shows that the incrementalized implementation is unaffected by the increasing number of roles in the system, which is in line with the `cost` annotations from Figure 1.3. In contrast, the straightforward implementation of `CheckAccess` is linear in the number of roles in the system; so are `CreateSession` and `DeleteSession`, albeit with a much smaller slope, as they occur only once per session, compared to the 1000 times of `CheckAccess`. The total running time of the straightforward version is linear in the number of roles, while the running time of the incrementalized version is constant. This improved asymptotic behavior leads to a practical speedup; with 100 roles, incrementalization improves the total running time from 0.94 to 0.37 seconds.

Figure 3.4(b) shows the results of a second experiment, where the number of roles in the system is fixed at 30 as the number of permissions per session varies. Again, the results conform to our expectations. The asymptotic cost of the incrementalized session creation and deletion increases with the number of permissions per session, while the cost of `CheckAccess` remains constant. The running time of all of the operations in the straightforward version are also asymptotically constant, although the practical cost of the straightforward version of `CheckAccess` is larger than that of the incrementalized version. Again, the time complexity of the incrementalized `CheckAccess` agrees with the results computed from the `cost` annotations from Figure 1.3.

Join. Our second example of generating efficient implementations from clear and modular specifications is a join operation, which can be written as a



(a) 100 permissions per session



(b) 30 roles

Figure 3.4: Running time of Core RBAC operations, 1000 repeats.

comprehension of the form:

```
{[x,y]: x in s, y in t | f(x)=g(y)}
```

Converting this to Python and adding some support code gives a program of 10 lines in length, corresponding to the following high-level program:

```
result = new set()
for x in s:
  for y in {y in t | f(x)=g(y)}:
    result.add([x,y])
```

This program contains a single expensive computation, the comprehension `{y in t | f(x)=g(y)}`. When incrementalized over updates to the `s` and `t` sets, the program expands to 24 lines in length.

We created two series of test data to evaluate the performance of the straightforward and incrementalized versions of join. In both series, the size of the two input sets was given as the independent variable `N`. The series differ in the size of the output. One series produces output of size $\mathtt{N}^2$, while the other produces no output at all, as would be the case with fully disjoint input. These two series let us explore the full limits of possible running times.

We ran both programs on both series of inputs. Figure 3.5 shows the running times. The straightforward program is always quadratic in running time, while the incrementalized program is quadratic or linear, depending on the size of the output. Thus, starting with a quadratic specification of join, we automatically obtained an incrementalized implementation that runs in time proportional to the size of the input and output; this is asymptotically optimal [107, 45].

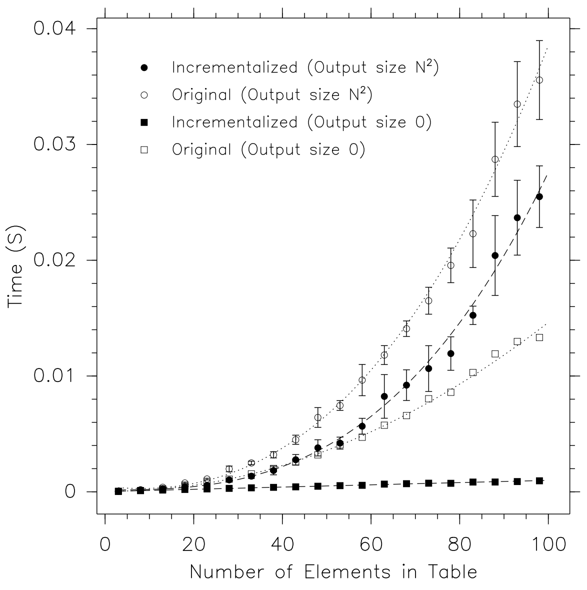


Figure 3.5: Running time of join.

3.2 Runtime invariant checking

Program safety, security, and general correctness properties depend on all kinds of invariants holding during program execution. Even though static analysis can verify many invariants, many important invariants are still too difficult to verify automatically using static analysis. Therefore, it is critical to use dynamic techniques to check during program execution that these invariants hold. This is known as *runtime invariant checking*. It is challenging for at least three reasons:

1. invariants that relate information at multiple program points are difficult to specify and to verify at any one point in the execution,

2. the runtime overhead from invariant checking must be minimized, and

3. imminent violations of critical invariants must be detected before they occur, and appropriate actions taken in response.

This section describes a general and powerful framework for efficient runtime invariant checking. The framework supports (1) declarative specification of arbitrary invariants using high-level queries, with easy use of information from any data in the execution, (2) powerful analysis and transformations for automatic generation of instrumentation for efficient incremental checking of invariants, and (3) convenient mechanisms for reporting errors, debugging, and taking preventive or remedial actions, as well as recording history data for use in queries. The transformations use InvTS as the backend.

We also describe a number of case studies that demonstrate the advantages of our framework and the effectiveness of our implementation. The implementation is for Python. The experiments include checking invariants about (1) abstract syntax tree (AST) transformations on programs of varying sizes between 400 and 16000 AST nodes, (2) Kerberos authentication used by a SMB client, and (3) a network protocol for distributing files in BitTorrent. All the invariants of interest can be expressed easily in our framework, and performance results show that our incremental checking scales well on large applications and complex invariants.

Much research has been done on runtime invariant checking, including a large variety of languages for specifying the invariants and methods for efficient instrumentation, as discussed in Section 3.5. To the best of our knowledge, no previous work both supports the generality of the kinds of invariants that our framework supports and achieves the efficiency that our implementation method achieves.

We now give an overview of our framework and describe the language for specifying invariants and actions; describe the transformations for incrementally checking the invariants; present our experiments; and discuss related work.

3.2.1 Framework

Invariants are expressed as boolean conditions involving variables quantified over collections. Violations of an invariant correspond to tuples containing values of those variables for which the condition is false. We formulate runtime invariant checking as evaluating queries that return sets of such tuples. The basic form of an invariant checking rule in our framework is

```
foreach ($v_1$ in $S_1$, ... , $v_k$ in $S_k$: condition):
    action
```

where S_1 through S_k are collections (sets, dicts and other collections that do not allow duplicates and that have constant-time membership tests). v_1 through v_k are quantified over sets S_1 through S_k, respectively. The set of tuples of values of v_1 through v_k such that *condition* holds is called the *query result*. *action* is a sequence of statements to be executed for each violation of the invariant, i.e., for each tuple in the query result.

For example, the following rule may be used to check that the `usage_count` field of each instance of the `File` class is non-negative:

```
foreach (o in extent(File): o.usage_count < 0):
   report("Error: File ", o, " has negative",
          " usage_count.")
   stop()
```

For every class C, `extent(`C`)` is a special set defined by our framework to contain the set of currently existing objects of type C. The `report` and `stop` functions are two functions in the subject programming language (Python): `report` takes any number of arguments and prints the concatenation of their string representations; `stop` stops the program and drops into a debugger, allowing the user to examine the state of the program at the point at which the invariant was violated.

While it is easy to see how to efficiently check simple invariants like the one above (by inserting checks at all assignments to the `usage_count` field), it becomes more difficult for even slightly more complex invariants. For example, consider a program that manipulates ASTs. We want to check that no node has an edge to itself. Assume that AST nodes are instances of the `Node` class, which has a `children` field. The invariant can be checked using the rule:

```
foreach (o in extent(Node): o in o.children):
   report("Error: ", o, " has a self-edge.")
   stop()
```

Checking this invariant efficiently is difficult, because aliasing implies that it can potentially be violated by any statement that adds an object to a collection, as in this scenario: `x=o.children; ...; x.add(o)`. Manually writing code to detect such bugs is tedious: one must intercept all calls to the `add` method of all instances of `set`, determine whether the target object equals the `children` field of some instance of `Node`, etc. In our framework, the user writes the simple rule above, and our system takes care of the rest, generating correct and efficient code for it.

Queries that involve multiple variables typically involve join conditions, which relate the values of the variables. For example, suppose the ASTs in the previous example should also satisfy the invariant that every node has at most one incoming edge. This can be checked using the rule:

```
foreach (n in extent(Node), m in extent(Node),
         c in extent(Node): c in n.children and
         c in m.children and n!=m):
   report("Error: ", c, "is a child of both ",
     m, " and ", n, ".")
   stop()
```

Again, it is easy to write this rule in our framework, but it is difficult to manually write code that efficiently checks this invariant at runtime, since this requires maintaining auxiliary data structures with information about edges, in addition to dealing with the aliasing issue discussed above.

Some invariants cannot be expressed using queries over extents and existing sets in the program. For example, consider a communication protocol. A query cannot refer to the set of all packets sent by the program, unless the program happens to maintain that set. It is not an extent, because packet objects are removed from the extent by garbage collection. To support such queries, our framework supports rules that add code throughout the program. This feature is similar to aspect-oriented programming, and it can be used to insert code that maintains additional sets.

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

Figure 3.6: General form of an invariant checking rule.

The general form of an invariant checking rule is shown in Figure 3.6. The syntax of the new clauses is taken from InvTL, where they are used in rules that describe how to maintain invariants; this is why we use *update* and *maint* as suggestive names for the code patterns in the `at` and `do` clauses, but they are not limited to matching updates and specifying maintenance code. The `at` clause contains a code pattern *update*, which may contain subject-language code and metavariables. As in InvTL, names of metavariables start with `$`. For each part of the code in the subject program that matches the *update* pattern in the `at` clause, if the *condition* in the `if` clause is satisfied, then the *declaration*s in the `de` (mnemonic for "declaration") clause are inserted into the program in the specified scope (while the `de` clause is usually used to declare and initialize variables, classes, or fields, it can be used to insert arbitrary code at a specified location) and the *maint* code in the `do` clause is inserted `before` or `after` the matched code, as specified, or, if `instead` is used in the `do` clause, the matched code is replaced with the code in the `do` clause. The condition in the `if` clause 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, or module.

Continuing the above example, the following rule could be used to check an invariant about packets that is expressed in terms of a set `$sent_packets` containing all sent packets (a specific example appears in Section 3.2.3). The metavariable `$sent_packets` gets instantiated with a fresh program variable when the program is transformed.

```
foreach (...: ... $sent_packets ...):
   report("Error : ...")
   stop()
de in global:
   $sent_packets=set()
at $x.send($packet)
if extends(type($x),socket)
do before:
   global $sent_packets
   $sent_packets.add($packet)
```

3.2.2 Generation of invariant rules

The straightforward way to implement the framework described above is to compute the result of every query from scratch at every program point. This is clearly correct, yet very slow, especially if the query involves large collections. A better way is to compute each query result at the program points that can update the result of the query. This is faster, yet still requires repeated evaluation of the query. A better approach is to efficiently maintain (i.e., update) the result of the query whenever a collection or object the query depends on changes.

This requires two steps: (1) generating maintenance code that properly maintains the query results in the face of updates to the data the query depends on, and, (2) applying the maintenance code at all places where the query result might change. The rest of the section uses "set" instead of "collection" as the method applies (with very minor modifications) to any collection that contains objects, does not allow duplicates, and has constant-time membership testing.

Step 1 is accomplished by compiling the query into an InvTS rule, which then transforms the subject program so that it incrementally maintains the query result.

Step 2 is performed by InvTS itself, as described in Chapter 2. To reiterate, InvTS inserts the maintenance code from step 1 at every location that updates the variables the query depends on. The straightforward way is to insert maintenance code at every statement in the program, preceded by a runtime check of whether the statement actually updates the data the query depends on. This slows down the transformed program even when no such updates occur, due to the evaluation of the runtime check at every statement. InvTS uses control-flow, data-flow, type, and alias information to evaluate as many of these checks as possible at compile time, to reduce the runtime overhead of maintaining the query result.

Generating maintenance code. As InvTS alone cannot generate the code to maintain a query result, we give a method that, for a class of queries, generates maintenance code (in the form of InvTS `at/if/de/do` clauses) that incrementally maintains the result of these queries.

We generate efficient maintenance code for queries of the form (v_1 `in` S_1, ..., v_k `in` S_k: *condition*), where *condition* is a conjunction and each conjunct is either (1) a join condition of the form e_1 *op* e_2, where *op* is `==`, `!=`, `in`, or `not in`, and e_i is v or $v.f$, where v is a variable and f is a field, or (2) a boolean expression whose value depends only on the objects bound to v_1, $\ldots$, v_k, the fields of these objects, and immutable objects.

Three kinds of updates can affect the result of a query: adding an object to a set, removing an object from a set, and changing the value of a field on an object. We decompose more complicated updates into these simple updates. We further simplify the problem by replacing field updates (for both scalar and set fields) with code that removes an object from all sets containing it, updates the field, and re-adds it to all sets. This transformation requires maintaining an auxiliary map from each object to the sets containing it.

With this simplification, the query result can increase only when an object is added to any of the sets S_1, $\ldots$, S_k, and the query result can decrease only when objects are removed from these sets. Since the action is executed only when the result set increases, this means that we only need to handle the addition case appropriately to update the query result. However, during removal we may need to update auxiliary maps.

Handling element addition. To handle addition of an object to a set, we run the query with the corresponding `v` variable bound to the object being added. We then generate statements corresponding to each of the clauses (enumeration, predicate, and join) in the query. The code is generated in the following order:

1. For a predicate with all variables bound, an if-statement checking the predicate is generated.

2. For an enumeration of the form v `in` S where v and S are both bound, an if-statement that performs a membership test is generated.

3. For a join condition with both variables bound, an if-statement that checks whether the join condition is satisfied is generated.

4. For an equality or set-membership join with exactly one variable bound, a for-statement that iterates over the entry corresponding to the bound variable in a hash-join map is generated.

5. For an enumeration where only S is unbound, a for-statement that iterates over the elements of S is generated.

If a clause does not match one of the conditions in this list, then it cannot be generated yet. Each generated for-statement binds a variable, which can cause statements to become generable or to rise in priority. As all variables can be bound through the *for* statement, eventually all clauses will be generated. The generated InvTS code has the form of additional `at`, `if`, `de`, and `do` clauses that, `at` each element addition, `do` the above-described maintenance.

Handling joins. For each join, we maintain a hashmap, which we call a hash-join auxiliary map. For example, for the join `v1.parent==v2.name`, if `v1` is bound, and `v2` iterates over `S2`, we introduce a hashmap with domain `S2` that maps `o.name` to `o`. Maintaining these mappings requires the generation of additional code which must be run in response to the addition and removal of elements of `S2` and changes to `o.name`. This code must be run before the maintenance code that handles element addition. Thus, either new `at/if/de/do` clauses are created, or existing ones are modified so that the new maintenance code is prepended to the appropriate `do` clauses.

Auxiliary clauses. The `at`, `if`, `de` and `do` clauses have the same syntax and meaning as in InvTS. Thus they are copied into the InvTS rule being generated.

3.2.3 Experiments

To demonstrate that our technique can efficiently verify invariants, we have applied it to invariants from multiple domains: abstract syntax tree transformations, authentication, and a file distribution protocol. For each invariant, we compare the performance of the program without any invariant checking; with invariants being checked incrementally using the method described in this paper; and with invariants checked in a non-incremental manner by re-evaluating the query from scratch each time an update occurs.

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

AST transformations

An abstract syntax tree (AST) should satisfy several invariants. For our first two experiments, we check that no AST node is its own child, and that each AST node is the child of at most one parent.

For these experiments, we apply InvTS to itself to create checked-InvTS, a version of InvTS that checks to ensure that program transformations do not violate the AST invariants. Checked-InvTS is then run with a rule-set that transforms subject programs into static single-assignment (SSA) form. Note that in this case, we are checking the correctness of checked-InvTS, rather than the programs it is applied to.

Not own child. Recall from Section 3.2.1 that the following rule detects violations of the invariant "a node is not a child of itself".

```
foreach (o in extent(Node): o in o.children):
   report("Error: ", o, " has a self-edge.")
   stop()
```

Figure 3.7 shows that checking this invariant causes a constant factor slowdown. The overhead is close to 70%. About half of this overhead is the cost of maintaining extents, while the other half is the cost of maintaining invariants.

We do not give the running time of the non-incremental instrumentation, as not even the smallest experiment was able to complete in the time limit of 20 minutes. Since the query is run each time an AST node is created or updated, the non-incremental version incurs an asymptotic slowdown. Incremental instrumentation eliminates this asymptotic penalty, rendering invariant checking practical.

No shared child. In an AST, no two parents may refer to the same child. The following rule checks for violations of this invariant:

```
foreach (n in extent(Node), m in extent(Node),
         c in extent(Node): c in n.children and
         c in m.children and n!=m):
   report("Error: ", c, "is a child of both ",
     m, " and ", n, ".")
   stop()
```

As this invariant contains multiple join conditions (`c in m.children`, `c in n.children`, `n!=m`), hash-join maps are used to evaluate it efficiently.

Figure 3.7 shows that incrementally checking this invariant increases the running time by less than 95%. In contrast, the non-incremental instrumentation would be cubic in the number of nodes currently alive in the program, as it iterates over three extents of nodes. This leads us to the estimate that, in the best case, the non-incrementally instrumented program is O($\#node^3$) worse than the uninstrumented one. It is not a surprise that all experiments

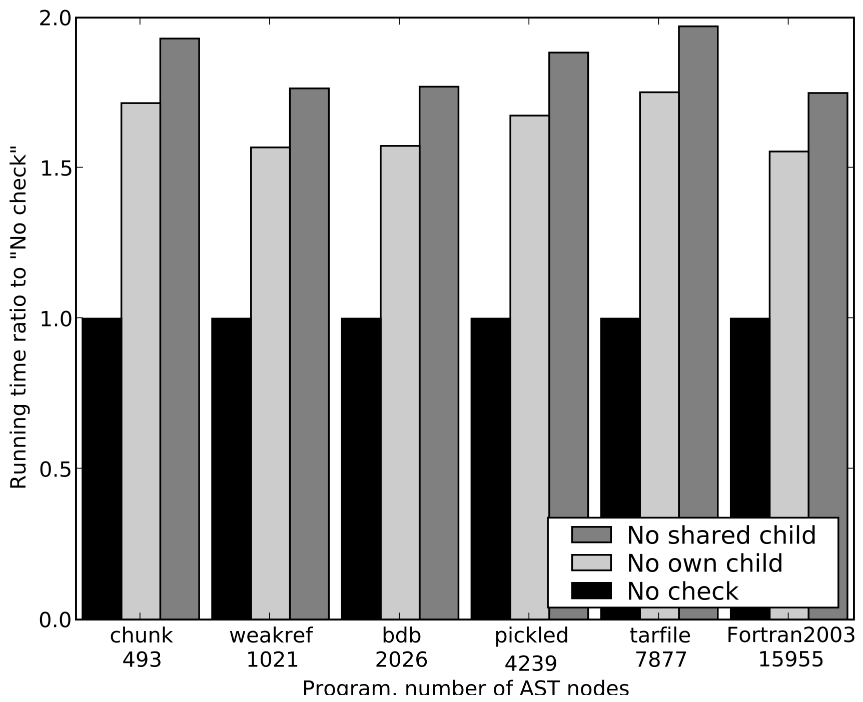


Figure 3.7: Running times of InvTS normalized to the running time of the non-instrumented version.

with non-incremental instrumentation timed out. When we manually introduced a bug that assigned the same child to multiple parents, checked-InvTS detected the violation.

Overall, these experiments show that verifying invariants at runtime can be efficient (with overhead smaller than 95%) for even 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 classes for which we maintain extents.

Authentication

We performed two experiments involving the Kerberos authentication used by pysmb, a SMB client written in Python. The first checks that all packets sent are authenticated; the second checks that authentication does not occur more frequently than necessary.

Require valid ticket. Our first experiment checks that we do not send packets to hosts that have an invalid Kerberos ticket associated with them.

This invariant needs to remain true until the packet is actually sent. To find violations of it, we keep a set of packets being sent, and report an error if a packet in the set is associated with an invalid ticket.

```
foreach (sp in $sending_packets,
         kt in extent(KerberosTicket):
         kt.invalid and kt.ip==sp.target_ip ):
   report("Sending ", sp, " with invalid ticket!")
   stop()
de in global:
   $sending_packets=set()
at $x.send($p):
if subclass(type($x),asyncore.dispatcher):
de in class type($x) in function handle_write($arg):
   if $arg in $sending_packets:
      $sending_packets.remove($arg)
do after:
   if $p not in $sending_packets:
      $sending_packets.append($p)
```

This rule tracks all sends of data over asynchronous sockets, and stops the program when a packet was sent to a server with an invalid Kerberos ticket. The `de` and `do` clauses work in the following manner: When a `send` method call is encountered, the packet being sent is added to the `$sending_packets` queue. It is removed from there once the packet is actually sent, which may not be necessarily immediate. This is detected by intercepting the `handle_write` callback in the class subclassing `asyncore.dispatcher`. This callback is called by Python when a packet is actually sent out over the given socket.

When we ran this on pysmb, while transferring a 10GB file over a 100Mbit connection, the average CPU load increased from 3.6% to 11.7%. The throughput remained the same, because the program was IO-bound in both cases. The increase is due to the join and the fact that many Kerberos tickets may match an IP address. A straightforward implementation increased CPU usage to 97%, and reduced the throughput of the program by 73%, as pysmb became CPU-bound. The times taken by the program to transfer the file were 1302 seconds for the uninstrumented version, 1351 seconds for the incrementally instrumented version, and 6321 seconds for the non-incrementally instrumented version.

Repeated authentication. It is inefficient for a program to request tickets from the Kerberos server long before the currently valid ticket times out. Thus, a useful invariant to check is that a successful authentication is not repeated until the resultant ticket is about to time out. A ticket times out when there was no activity relating to that ticket for 300 seconds, i.e., no data was sent

	No check	Incremental	No type anal.	No alias anal.	Non-incr.
pysmb - Require valid ticket	3.6% (1302s)	11.7% (1351s)	19.7% (1819s)	14.1% (1601s)	97.3% (6321s)
pysmb - Repeated authentication	3.6% (1302s)	17.9% (1535s)	31.7% (2011s)	23.3% (1943s)	96.9% (8750s)
BitTorrent - No duplicate data	28.3% (1771s)	36.1% (1779s)	63.8% (1790s)	36.3% (1830s)	99.8% (3210s)
BitTorrent - No packet modification	2.7% (1783s)	3.3% (1687s)	3.9% (1763s)	3.4% (1805s)	93.1% (1801s)
InvTS - No shared child	13s	25s	349s	25s	>1200s
InvTS - No own child	13s	21s	312s	26s	>1200s

Table 3.1: CPU utilization (if IO-bound) and wall time taken for experiments under differing optimizations.

to the host the ticket was issued for the last 300 seconds. Thus the invariant is: there are no two valid tickets such that they are both referring to the same host, are both valid, and are much less than `timeout` (i.e., 300 seconds) apart. We define much less as 10 seconds less, as the MIT Kerberos client requests a new ticket 10 seconds before the current one times out. To verify this invariant, we need to keep track of Kerberos tickets and of SMB activity.

The invariant is expressed using a nested query, with the inner query computing the latest packet sent to a given host, and the outer query doing a join on all pairs of currently existing Kerberos tickets. The max aggregate is maintained using a heap.

```
foreach (k_old in extent(KerberosTicket) ,
         k_new in extent(KerberosTicket):
         k_old.valid and k_new.valid and
         k_old.issue_time<k_new.issue_time and
         k_old.ip==k_new.ip and
         k_new.issue_time-max([p.time
            for p in $sent_packets
            if p.target_ip==k_new.ip and
               p.time < k_new.issue_time])
         < 300-10):
   report ("Reauthenticated to host ", k_new.ip )
   stop()
de in global:
   $sent_packets=set()
at $x.send($p)
if type($x)==asyncore.dispatcher
do after:
   $sent_packets.add($p)
```

When run on pysmb, while transferring a 10GB file over a 100Mbit connection, the average CPU load increased from 3.6% to 17.9%, mainly due to the need to maintain a heap per IP address, and an additional join over the previous example. Using specific domain knowledge, the heap could be avoided: we could just keep track of the latest packet sent to each IP address. This works because time is monotonic. A rule modified in such a way is less easily adapted towards other uses, though. Note that even with the maintenance of the heap, the instrumented program is still IO bound, not CPU bound. Checking invariants in a non-incremental manner makes it CPU bound: it results in a 96.9% CPU load, and the running time increases from 1302 to 8750 seconds.

The pysmb examples show that instrumenting complex programs in ways not anticipated by their creators is easily done with our framework due to the ability to specify complex program transformations, such as maintaining the set of sent packets, or the set of packets waiting to be sent. It also demonstrates that complex conditions, including nested queries, are supported by this framework, and their use does not cause excessive overhead.

File distribution protocol

BitTorrent (http://download.bittorrent.com/dl/) is a peer-to-peer file distribution protocol. When multiple peers download the same file concurrently, they can relay data to each other, making it possible for the file source to support very large numbers of downloaders with only a modest increase in its load. Each peer downloads chunks of a file from (likely different) peers, and then reassembles the original file from the chunks. The BitTorrent protocol is relatively complex, so we use our method to instrument an implementation and check it for potential errors.

No duplicate data. Receiving the same piece of data from two sources too often may mean that the client is using bandwidth inefficiently. We check for this using a rule that detects when the same data is received from two or more distinct sources (identified by IP address), and logs the event without stopping the program. The log could be analyzed later to determine whether the duplication is due to a bug or misconfiguration.

```
foreach (p1 in $in_queue, p2 in $in_queue:
         p1.source_ip!=p2.source_ip and
         p1.payload==p2.payload):
   report("Receiving same data from peers ",
     p1.source_ip, " and ", p2.source_ip)
de in global:
   # A queue of incoming packets.
   # It supports O(1) membership tests,
   # holding at most 100000 packets
   $in_queue=queue(max_length=100000)
at $x.type=$s
if $s=="incoming" and type($x)==Packet
do after:
   if $x not in $in_queue:
      $in_queue.append($x)
```

Experiments involved receiving a 10GB file from 30 peers, over a 100Mbit connection. We measured CPU load to determine the impact of the runtime checking. The average CPU load increased from 28.3% for the original program to 36.1% for the instrumented program. The small increase is due to the high selectivity of the `p1.payload==p2.payload` join condition. Just like with pysmb, both versions of the program are IO-bound.

No packet modification in transit. To verify that the correct data is being sent between peers, we check the following invariant: a packet sent from one peer must be received by another peer without a change in the payload.

We check this invariant by creating a server to which peers send summaries of the packets they send and receive. These packets are put into a set on the server. We write a query that detects when packets of the same chunk have a different payload, by comparing the MD5 hashes of the payloads.

The server maintains a set `rec_set` containing all packets sent and received by BitTorrent peers. The following rule checks the invariant:

```
foreach ($from in self.rec_set, $to in self.rec_set:
         $from!=$to and $from.source!=None and
         $from.target!=None and
         $from.source==$to.source and
         $from.target==$to.target and
         $from.chunk==$to.chunk and
         $from.chunk!=None and
         $from.sent and $to.received and
         $from.md5!=$to.md5 and $from.md5!=None):
   report ("Packet sent from ", $from.source,
     " to ", $from.target, " changed in transit!")
   stop()
```

We use two InvTS rules to modify the BitTorrent program to send the information needed for invariant verification to the server. The rules state that a socket should be opened to the server once per program, and that anytime a packet is written to any socket, or read from any socket, the packet (minus the body) should be sent to the server. The rule for handling `send` is the same as the rule below for handling `receive`, with `receive` replaced with `send`.

```
at $x.receive($p)
if type($x)==asyncore.dispatcher
de in global:
   import socket
   #Open a socket to server on 192.168.17.46:636
   $check_socket=socket.open_udp(192.168.17.46,636)
   in global in function(myreceive(socket,packet)):
      global $check_socket
      # For efficiency, do not sent the payload
      $body=packet.body
      $arg.body=None
      $check_socket.send(packet)
      packet.body=$body
```

```
do instead:
   myreceive($x, $p)
```

After applying the query and rules to the BitTorrent client and our server, we benchmarked the CPU utilization of the clients and the server (which were running on the same computer). With 5 BitTorrent clients and the server running, the CPU utilization increased from 73 to 78 percent. When the clients were measured in isolation, the CPU utilization of a single client (with the other 4 clients and the server running on another system) was 11%, vs. 10% for the untransformed client. The server, when run on the test machine (with the 5 clients running on a different machine) utilized 3.3% of the CPU with the instrumentation enabled, versus 2.7% with no instrumentation.

On a reliable connection we found no problems. When we simulated a bad connection by randomly injecting changes into some packets, we found the errors, before the BitTorrent error detection algorithm, which operates on bigger chunks.

Effect of optimizations. Table 3.1 shows the CPU utilizations and running times of the pysmb and BitTorrent examples under different implementation options. It is easy to see that the non-incremental implementation is far worse than any other version. Disabling type or alias analysis also produces a noticeable slowdown.

3.3 Query-based debugging

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.

We describe 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.

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 the cause of a previously noticed non-injected bug in the FTP client.

Query-based debugging has been studied for at least a decade [66] and has received increased attention in recent years [67, 82, 74, 109]. 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.

3.3.1 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 3.8.

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

Figure 3.8: 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 on only the objects in the containers (S_1, …, S_k) iterated over by the query , 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 `de`, `at`, `if`, and `do` clauses have the same semantics as in InvTL.

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 3.9 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 ",m)
   stop()
```

Figure 3.9: A child's `parent` field must point to its parent.

As before, 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 *query 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 the following statements that form a cycle: `x=o.children; ...; o.add(x)`. 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 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 [56]. 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 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, 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 3.10 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` clause). It is removed from the set im-

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

mediately after (`do after` clause) the `cwd` call returns. The same is done for `ls` and other FTP commands. `$exec_commands` is a metavariable; it will be instantiated with a fresh program variable, whose value will be set to a new empty set (`de` clause). `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 3.11 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 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 special metavariable that expands to an object that identifies the statement being transformed. `Update` stores

```
foreach (c in $C, i in $I: i.value != None and c.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 3.11: A rule that helps determine the cause of an uncaught exception.

two fields: `locId` and `value` store an object identifying an lvalue and an arbitrary value, respectively. Only `locId` is used for comparing instances of `Update`. Thus, `$C.discard($obj)` removes the entry with the same `locId` as `$obj` from `$C`, and `$C.add($obj)` adds to it `$obj` with the new `value`. `locId` is a also function that generates an identifier that uniquely identifies an lvalue. Updates to `$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,'$var',cond)` is a special function that finds a set M of minimal subexpressions of `$e` such that `cond` is true for each such subexpression, and, for each elemement of M, binds `$var` to that element. Note that if there are no updates to either `$L` or `$R`, then code to maintain `$C` or `$I` 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, using the transformations described in the previous section. 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, which then transforms the subject program so that it incrementally maintains the query result. The resulting rule looks like a rule in Figure 3.8, 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".

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 3.3.2 (especially Figure 3.12).

3.3.2 Experiments

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.

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.

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 3.9 stops the program when an element that violates that property is found. Figure 3.12 shows that the overhead of running the incrementally instrumented program in qbdPy is 68%. 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 3.12 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.

A Python FTP client. We found the cause of a previously noticed bug in a program that downloads directories from multiple machines [54]. This bug involves 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 3.10 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 3.12 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 3.2.

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.

	Running time					Instrumentation + running time	
	No debug.	All anal.	No type	No alias	No anal.	All anal.	No anal.
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 3.2: Time taken for experiments under differing optimizations.

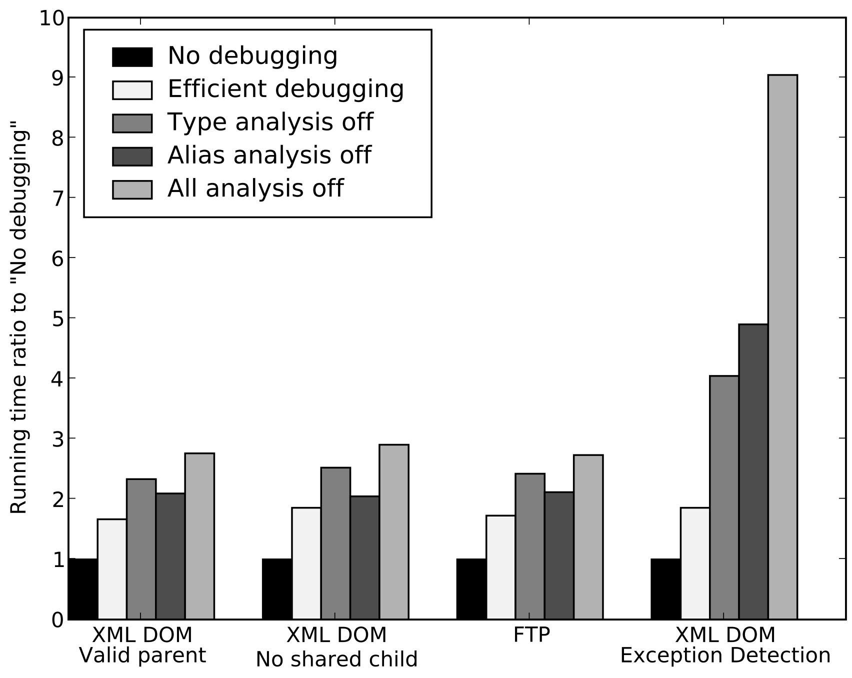


Figure 3.12: Running times of applications in qbdPy, normalized to the running times of the applications outside of qbdPy.

Automated determination of causes of exceptions. In the final case study in Section 3.3.1, Figure 3.11, 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 IndexError into lxml, and ran it after applying this debugging rule to it. The result, as can be seen in Figure 3.12 and Table 3.2, 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 3.12, explains the high performance of the query.

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 3.2 presents the results. None of the programs take longer than 1 minute to instrument with all analyses turned on, and none took less than 20 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.

3.4 Other applications

Instrumentation. File Access Profiling looks for a specific file access pattern. It demonstrates that one can easily express access patterns of interest in InvTL, much like in an aspect-oriented programming system. Reference Counting emulates a reference-counting garbage collector, by incrementally maintaining reference counts. It uses static analysis in InvTS to avoid keeping reference counts for values of primitive types, and thus avoid severe performance degradation usually associated with explicit reference counting. Memory Coverage instruments `ViM`, a text editor, to intercept all calls to `malloc` in order to monitor memory access patterns. It is challenging because `malloc` may be called indirectly through function pointers aliased to `malloc`.

Refactoring. InvTS Refactoring was a real experience in the implementation of InvTS. The goal was to rewrite InvTS, factoring out the Python language module from InvTS core to form InvTS/py. The rules mainly did variable and class renaming and method extraction, and turned a tedious manual task into an easy one. Variable Renaming is for C and involves small rules that rename only local, only global, only static, or only global static variables. These rules are easy to write because InvTL takes scope of variables into account. Implementing such renaming using, say, Perl scripts, would be significantly harder and very error-prone.

Other transformations. InvTS/py and InvTS/c Test Suites consist of miscellaneous transformations where transformed programs are checked automatically against known results for testing purposes. The transformations insert code that prints certain strings when certain code patterns are matched. These strings are then counted for simple tests.

3.5 Related work

This section touches four areas: specification and implementation of Core RBAC, runtime invariant verification [21], incremental query result maintenance, and query-based debugging.

Core RBAC. RBAC has been specified formally and precisely in a set-based specification language, Z [52, 97], in the ANSI standard [3, 33]. The standard defined and maintained incrementally several additional maps, in addition to or in place of, the relations defined in Section 3.1.1, which appears to be caused by efficiency concerns but complicates the specification unnecessarily. That specification was debugged and simplified in [70], resulting in the specification used in this work. There are many implementations of RBAC (e.g., [113]), but we are not aware of any that provides precise complexity guarantees.

Runtime invariant checking. There are several systems for runtime checking of temporal properties. These include Java-MaC [58], JPAX [49], JNuke [6], and EAGLE [11]. These systems express the properties in a linear temporal logic (LTL) or related rule languages.

Our system does not support writing invariants in LTL, although, as our system supports comprehensions, extents, and joins, a subset of LTL can be emulated. The pysmb example does so by maintaining history and specifying queries over it. While this may incur a performance penalty compared to systems specifically designed to test LTL-based invariants, it is not a very significant performance penalty (As seen in Section 3.2.3, the overhead is consistently under 100%).

The category into which our system fits best is tools that use a side-effect free subset of their subject language, extended with various operators such as quantifiers or set operations, to specify invariants. Such invariant specification languages include JML [64], Spec# [10], and Jahob [61]. For JML and Spec#, there are tools that allow the user to combine/compile an invariant and a subject program into a compiled program that, at runtime, checks whether the specified invariant holds. These tools include Boogie [9] for Spec# and jmlc [17], jass [12], jmle [60], and DITTO [91] for JML. A runtime verifier for Jahob is under development [112].

Spec# does not support comprehensions[112] or extents. As such, it cannot easily encode the invariants we wish to verify. JML supports set comprehensions, quantifiers, and other features. It does not natively support extents [65]. Jahob supports both comprehensions and extents (as a subset of the `AliveVariables` set). The language presented in this paper supports both set comprehensions and extents. It is worth noting that support for extents is difficult to emulate without support for liveness testing, because garbage collection must be taken into account.

The JML compilers jmlc, jmle, and jass all support a large subset of JML, including comprehensions. But, they evaluate comprehensions in a straightforward manner, by recomputing them whenever they are encountered. In contrast, our system incrementally maintains the value of set comprehensions. DITTO provides incremental maintenance of some JML expressions, but it does not incrementally maintain set comprehensions [91].

Incremental query result maintenance. JQL [109] extends Java to support both comprehensions and extents, to support querying over collections. Recent work on JQL adds incremental maintenance of JQL queries in the face of updates to the data they depend on. The fact that our system is designed with only invariant verification in mind allows us to more efficiently maintain invariants. For example, it is easier for us to handle removal of elements from the sets that the query depends on. We support a marginally 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`. Also, the `at` and `de` clauses allow us to do program transformations that maintain datastructures that would be unavailable to a pure query language, such as a set of all previously sent packets.

Potanin et al. [82] query snapshots of object graphs, but perform the queries non-incrementally. PQL [74] queries over past states of the program, but not over extents. It uses BDDs to compute query results, but not incrementally.

Acar et al. [47, 1] perform self-adjusting computation which is a combination of change propagation and memoization, for ML and C, which works quite well for recursive algorithms. Unfortunately, their method (1) does not supports automatically incrementalizing any set comprehensions, and (2) suffers from significant overhead — factors of up 100 for ML, and of around 6-10 for C [1].

Query-based debugging. Query-based debugging has recently received a great deal of attention [67, 82, 74, 109], 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 language is an extension of the query-based debugging language by Lencivicius et al. [67]. The method in [67] 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. [82] allows querying snapshots of object graphs, and also performs these queries non-incrementally. PQL [74] allows queries over past states of the program, but not over extents. It uses BDDs to efficiently compute the query results. PTQL/PARTIQLE [37] 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 [109] 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 [108] 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 language for defining pointcuts. The pointcut language of AspectJ[56] is somewhat limited; other proposals [2, 85] are more expressive. In particular, these proposals allow advice to execute based on the history of program execution.

The goals qbdPy 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).

Chapter 4

Alias analysis for update detection

Update detection. To apply an invariant rule in a coordinated fashion, InvTS must detect all updates to the parameters of a query, even under object aliasing. It is generally impossible to do this precisely at compile-time. Thus, when InvTS knows at compile-time that a statement may update a parameter of the query, it inserts code that checks at runtime whether that statement updates that parameter of the query. As such runtime checks are expensive, InvTS attempts to introduce as few of these checks as possible — it only introduces a runtime check at a statement if it can say, after static analysis, that the statement *m*ay update the value of the parameter but not that the statement must update the value of the parameter.

Thus, update detection boils down to answering two questions:

(1) Given a parameter p and an update u to a variable v, *m*ay u update the value of p? The answer is yes if:

1. the query that uses p is reachable from u, and
2. v and p may be aliased to one another.

(2) On the other hand, must u update the value of p? The answer is yes if:

1. u *m*ay update the value of p, and
2. v may not be aliased to any variable other than p.

Given control-flow, type, and may-alias information for the program being transformed, it is trivial to answer these questions. For C, there exist a plethora of algorithms for computing control-flow, type, and may-alias information; unfortunately, this is not the case for programs written in a dynamic language such as Python, a language that InvTS supports.

To overcome this shortcoming, this chapter describes the development and experimental evaluation of a may-alias analysis for a full dynamic object-oriented language. As the may-alias analysis algorithm requires control-flow and type information about the program being analyzed, this chapter also presents an algorithm for obtaining this information. These analyses allow us to perform update detection for Python programs, as described above.

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 [83]. 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 chapter 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 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 [44] 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 and applications that use Psyco, a just-in-time compiler that does specialization [84]. 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 4.3. 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 by showing that they are essential for the analysis required by InvTS. Consider the following simple example that contains updates to collections and is typical in dynamic languages:

```
#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`. As mentioned before, 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, 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.

4.1 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 chapter, *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.

4.1.1 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 $list_{all}[t_1, ..., t_n]$ for lists of known length n and element types t_1 through t_n 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 $dict_{all}[(kt_1, vt_1), ..., (kt_n, vt_n)]$ 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 t_1 is a subtype of a type t_2 iff all components of t_1 are subtypes of the corresponding components of t_2.

Any set $\{t_1, \ldots, t_n\}$ 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.

Analysis and refinement. Our algorithm does the Analysis step below to infer types and construct a CFG, and then does the Refinement step to specialize the program constructed based on the types inferred; the resulting program is then analyzed again to yield more precise types and more refined control flows, and is analyzed incrementally. Our overall algorithm repeats the two steps until either a fixed-point is reached, so 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 4.2.3 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 4.2.4 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 we bound the set of types considered and the number of refinements by constants.

4.1.2 Abstract interpretation

In abstract interpretation, the collecting semantics of a program is expressed as a least fixed-point of a set of equations. The equations are solved over an abstract domain chosen based on desired precision and cost. The equations are solved iteratively; that is, successive approximations of the solution are computed until they converge to a least fixed-point [38]. We describe both the abstract domain, and how we interpret the various language constructs.

While performing abstract interpretation, we generate the CFG for the entire program. Unless specified otherwise, we add an edge from the last node we interpreted to the node we are interpreting currently; exceptions are described explicitly. We use AST nodes as CFG nodes, and thus sometimes omit the AST or CFG qualifier for brevity.

$$
\begin{array}{rcl}
t & := & \mathit{Int} \mid \mathit{Float} \mid \mathit{Bool} \mid \mathit{String} \mid \mathit{List} \mid \mathit{Tuple} \mid \\
 & & \mathit{Set} \mid \mathit{Dict} \mid \mathit{Module} \mid \mathit{Class} \mid \mathit{Instance} \mid \mathit{Func} \mid \\
 & & \mathit{Method} \mid \mathit{Union} \mid \mathit{none} \mid \mathit{bot} \mid \mathit{top} \\
\mathit{Int} & := & \mathit{int} \mid \mathit{int}_{val}(\texttt{n}) \mid \mathit{int}_{non\_neg} \mid \mathit{int}_{ran}(\texttt{n}, \texttt{n}) \\
\mathit{Float} & := & \mathit{float} \mid \mathit{float}_{val}(\texttt{f}) \mid \mathit{float}_{ran}(\texttt{f}, \texttt{f}) \\
\mathit{Bool} & := & \mathit{bool} \mid \mathit{bool}_{val}(\texttt{true}) \mid \mathit{bool}_{val}(\texttt{false}) \\
\mathit{Char} & := & \mathit{char} \mid \mathit{char}_{val}(\texttt{c}) \mid \\
 & & \mathit{union}((\mathit{char} \mid \mathit{char}_{val}(\texttt{c})) \quad (,(\mathit{char} \mid \mathit{char}_{val}(\texttt{c})))*) \\
\mathit{String} & := & \mathit{string} \mid \mathit{string}_{all}[(\mathit{Char} \;\; (\,,\mathit{Char} \,) * \;\;)?] \mid \\
 & & \mathit{string}_{len}(\texttt{n}) \mid \mathit{string}_{elem}(\mathit{Char}) \\
\mathit{List} & := & \mathit{list} \mid \mathit{list}_{all}[(t \;\; (\,,t \,) * \;\;)?] \mid \mathit{list}_{len}(\texttt{n}) \mid \mathit{list}_{elem}(t) \\
\mathit{Tuple} & := & \mathit{tuple} \mid \mathit{tuple}_{all}[(t \;\; (\,,t \,) * \;\;)?] \mid \mathit{tuple}_{len}(\texttt{n}) \mid \mathit{tuple}_{elem}(t) \\
\mathit{Set} & := & \mathit{set} \mid \mathit{set}_{all}[(t \;\; (\,,t \,) * \;\;)?] \mid \mathit{set}_{len}(\texttt{n}) \mid \mathit{set}_{elem}(t) \\
\mathit{Key} & := & t \\
\mathit{Value} & := & t \\
\mathit{Dict} & := & \mathit{dict} \mid \mathit{dict}_{size\_key}[\texttt{n}, \mathit{Key}] \mid \mathit{dict}_{size\_val}[\texttt{n}, \mathit{Value}] \mid \\
 & & \mathit{dict}_{all}[((\mathit{Key}, \mathit{Value})(,(\mathit{Key}, \mathit{Value}))*)?] \mid \\
 & & \mathit{dict}_{len}(\texttt{n}) \mid \mathit{dict}_{elem}(\mathit{Key}, \mathit{Value}) \mid \\
\mathit{Fields} & := & [(\; (\texttt{name}, t) \quad (\; , \; (\texttt{name}, t) \;) * \;\;)?] \\
\mathit{Module} & := & \mathit{module}(\texttt{name}?, \mathit{Fields}?, \texttt{ASTid}?) \\
\mathit{Parents} & := & [(\; \mathit{Class} \quad (\; , \; \mathit{Class}) * \;\;)?] \\
\mathit{Class} & := & \mathit{class}(\texttt{name}?, \mathit{Parents}?, \mathit{Fields}?, \texttt{ASTid}?) \\
\mathit{Instance} & := & \mathit{instance}(\mathit{Class}?, \mathit{Fields}?) \\
\mathit{Params} & := & [(\; (\texttt{name}, t) \quad (\; , \; (\texttt{name}, t) \;) * \;\;)?] \\
\mathit{Freevars} & := & [(\; (\texttt{name}, t) \quad (\; , \; (\texttt{name}, t) \;) * \;\;)?] \\
\mathit{Return} & := & t \\
\mathit{Func} & := & \mathit{function}(\texttt{name}?, \mathit{Params}?, \mathit{Freevars}?, \mathit{Return}?, \texttt{ASTid}?) \\
\mathit{Method} & := & \mathit{method}(\mathit{Instance}, \mathit{Func}?) \\
\mathit{Union} & := & \mathit{union}(\; t \;\; (\,,t \,) * \;\;)
\end{array}
$$

Figure 4.1: Grammar for precise type t. Abstract interpretation domain is t. `n`: integer; `f`: floating-point; `c`: character ; `name`: identifier; `ASTid`: AST node identifier; suffix ? indicates that the clause may be omitted; suffix $*$ denotes that the clause may appear zero or more times.

Abstract domain. The abstract domain is the precise types described in Section 4.1.1. A grammar for the precise types is given in Figure 4.1.

Join. We define the join operator, $\sqcup$, such that $t_1 \sqcup t_2$ is the minimal supertype of types t_1 and t_2. The minimal supertype of two types is defined in Section 4.1.1.

Widening. In abstract interpretation, successive approximations of the solution are computed until they converge to a least fixed-point. However, if in a particular abstract domains (such as intervals) such chains of approximations can be very long or infinite, abstract interpretation uses an extrapolation technique called *widening* [25].

Widening attempts to predict the fixed-point based on the sequence of approximations computed on earlier iterations of the analysis. Typically, widening degrades the precision of the analysis, i.e., the obtained solution is a fixed-point, but not necessarily the least fixed-point [38].

We introduce an operation on a type called "generalization" that is equivalent to applying a widening operator to that type, in that it returns a more general type of the given type: generalization takes a type t, and outputs another type t', such that t' is a supertype of t, $t \neq t'$, and the size of t' is smaller than the size of t, where size is as defined in Section 4.1.1.

Abstract interpretation of Python constructs. We describe how we analyze all Python constructs. We use *type*(`expr`) to denote the type of `expr`, and *node*(`expr`) to denote the AST/CFG node representing `expr`.

Constant. We consider nodes that contain literal integers, floats, `true` or `false`, and strings to be constants. When interpreting a constant, we set the type of the node to the corresponding precise type, such as int_{val}(`n`), where `n` is the integer literal represented by the node.

Variable. We look up the type of the variable.

Field access `expr.fieldname`. We first determine *type*(`expr`), then, if *type*(`expr`) is a *Union*, for each element t of the union, if t is a *Class*, *Instance*, or *Module* type, we look up the type of the field `fieldname` in t, and set the type of the field access expression to the join of the resulting types from all elements of the union. If *type*(`expr`) is not a *Union*, we treat it as if it were a *union*(*type*(`expr`)).

`list` and `dict` special forms. A list special form is `[`$\mathtt{e}_1$`, `$\mathtt{e}_2$`, ..., `$\mathtt{e}_k$`]`. This creates a list of k elements, with the values of the expressions $\mathtt{e}_1, \ldots, \mathtt{e}_k$ being the elements of the list. A `dict` special form is `{`$\mathtt{e}_1$`:`$\mathtt{e}_2$`, `$\mathtt{e}_3$`:`$\mathtt{e}_4$`, ..., `$\mathtt{e}_{k-1}$`:`$\mathtt{e}_k$`}`. This creates a `dict` (a map) that maps $\mathtt{e}_1$ to $\mathtt{e}_2$, $\mathtt{e}_3$ to $\mathtt{e}_4$, etc. We interpret the node n representing the special form by first interpreting $\mathtt{e}_1, \mathtt{e}_2, \ldots, \mathtt{e}_k$, and determining their types. Once the types are determined, the type of the list or dict special form is, respectively, $list_{all}[t_1, t_2, \ldots, t_k]$ and $dict_{all}[(t_1, t_2), (t_3, t_4), \ldots, (t_{k-1}, t_k)]$, where t_i is *type*($\mathtt{e}_i$). We construct the CFG by first connecting the predecessor nodes of n to *node*($\mathtt{e}_1$), then connecting *node*($\mathtt{e}_1$) to *node*($\mathtt{e}_2$), ..., *node*($\mathtt{e}_k$). Finally, we connect *node*($\mathtt{e}_k$) to n.

Assignment. Assignment has three forms: basic assignment, field assignment, and `setitem` assignment.

Basic assignment `var=expr` assigns the value of `expr` to variable `var`. Basic assignments are strong updates. Thus, we create a new variable with the fully qualified variable name augmented by the `ASTid` of *node*(`var=expr`), and the type of the newly created variable set to *type*(`expr`). Consequently, if *node*(`var=expr`) is in function `F`, module `M`, and has `ASTid id`, then variable `var` has the full name $\texttt{M::F::var}_{id}$. Further uses of `var` in `F`, until another basic assignment to `var` is encountered, will refer to (and update the type of, if necessary) the variable that has the full name $\texttt{M::F::var}_{id}$. We also set *type*(`var=expr`) to *type*(`expr`).

Field assignment `expr1.f=expr2` assigns the value of `expr2` to either the class, class instance, or module referred to by `expr1`. If *type*(`expr1`) is a *union* type, then, for each type t in the *union*, we perform the following:

- if t is a $Module$, we replace t with t_{new}, where t_{new} is derived from t by, if `f` is in $Fields$, replacing the type of `f` from $Fields$ with *type*(`f`)$\sqcup$*type*(`expr2`), or if it is not in $Fields$, adding `f` to $Fields$ and setting its type to *type*(`expr2`);
- if t is a $Class$, we replace t with t_{new}, where t_{new} is derived from t by looking up *type*(`f`) first in t, then in the types in $Parents$ of t, in order of inheritance. If *type*(`f`) was successfully looked up, we add `f` to $Fields$ of t, and set *type*(`f`) to *type*(`f`)$\sqcup$*type*(`expr2`). Otherwise, we add `f` to $Fields$, and set *type*(`f`) to *type*(`expr2`);
- if t is an $Instance$, we do the same as for $Class$, except we first look up the type of `f` in the $Fields$ of t, then in the $Class$ of t, and only then in the $Parents$ of the $Class$.

If *type*(`expr1`) is not a $Union$, but a $Module$, $Class$, or $Instance$, we perform as if *type*(`expr2`) was the only element of the $Union$, but instead of doing weak updates by setting the type of `f` to *type*(`f`) $\sqcup$ *type*(`expr2`), we perform a strong update and set the type of `f` to *type*(`expr2`). Finally, if *type*(`expr1`) is *bot*, we replace it with $union(C, I, M)$, where C, I, and M are $Class$, $Instance$, and $Module$ types with unknown names, AST ids, etc., but with `f` added to $Fields$, and `f`'s type set to *type*(`expr2`).

`setitem` assignment `expr1[expr2]=expr3` is equivalent to a method call of the form `expr1.__setitem__(expr2,expr3)`. It is treated as method call on `expr1`, but with the optimization that when *type*(`expr1`) is either a *List*, *Dict*, or one of their subtypes, it is treated as field assignment `expr1.expr2=expr3`. This lets us avoid interpreting the function call for these very common operations.

Overloadable operator. In Python, most operators, including augmented assignments (`+=`, `*=`, etc.), are overloadable. Notable exceptions include the `=` (assignment) and `.` (field access) operators. For the built-in types (*Int*, *List*, *String*, ...) we provide special functions that specify the behaviour of applicable operators in terms of type behaviour based on their operands. For the rest of the classes in the standard library, we infer types based on analysis of Python code that provides a minimal implementation of the operators for the respective classes, provided by us. When an operator is encountered, based on the types of its arguments, we determine which functions provide its implementation, and for each of them, either a call to that function is performed (as described in function and method calls) if the function is a reference or complete Python implementation, or a special function is executed that returns the type of the result of the operator.

Branching and looping constructs: `if`, `for`, `while`, and list comprehensions. For `if`, the condition is interpreted first to obtain its type, and then, unless its type is $bool_{val}$(`true`) or $bool_{val}$(`false`), the entry points of the true and false branches are added to the worklist of nodes to be interpreted. The true branch is first interpreted, and then the false branch, if it exists, is interpreted. CFG edges are added from the condition to the first nodes of the true and false blocks, and from the last nodes of the true and false blocks to the node immediately following the `if` statement. `for` and `while` statements are handled in a similar manner, except the body and condition are interpreted in a loop until a fixed-point of the condition's type and the types of all variables inside the loop body is reached. We treat list comprehensions as a (potentially nested) combination of `if` and `for` statements that accumulates the result of the comprehension in a temporary variable, and returns the variable.

Function, constructor, and method calls. In Python, function calls and constructor calls are the same syntactically: `expr(params)`, where `params` is optionally a comma-separated list of parameters, optionally followed by list and keyword arguments. To determine what call to make, we first compute the type of `expr`. Assuming *type*(`expr`) is a *Union* type, then for every element t of it, we perform the following:

- If t is a *Func* or *Method* type, we perform the function or method call. As calls can be nested, we maintain a stack of current call sites. The *Func* or *Method* type is used to determine the parameters to the function, the free and default parameters, and the AST node of the function. Once done, the analysis pushes the call site node onto the stack, assigns the types of the formal parameters of the function to their local names, and starts analyzing the function body from its entry point. All `return` statements encountered in the function are treated as weak assignments to the special return variable, whose type is returned when the analysis reaches the function end. Then, the call site node is popped from the stack, and, if applicable, the return type of the function is assigned to the appropriate variable.

All of the above happens if a function with the same name and same types of all parameters and globals has not been analyzed before. Otherwise, the call site node is not pushed onto the stack, and the function is not analyzed again, and instead, the return type of the previous call is looked up, and is returned by the function. Method calls are analyzed similarly except that we set the type of their first formal parameter to their *Instance* type. CFG edges are created appropriately: from the call site to the entry point of the function, throughout the function as usual, from the `return` statements to the assignment of the return variable, and from all assignments of the return variable to the node following the call site.

- If t is a *Class* type, then the call is a constructor call. We first determine the class of the object being constructed. This involves checking whether the *Class* being constructed uses a metaclass: we look up whether the *Class* has static field `__metaclass__`. If it does, and the type of that field, say called m, is an *Instance*, and m contains a static field `__new__` whose type is a *Method*, then we interpret the call to m`.__new__`, and if its return type is a *Class* type, we invoke the constructor of that return type. Otherwise, we invoke the constructor of t. Next, we create an *Instance* type t_2 based on the *Class* type obtained above, with an empty *Fields* parameter. Then we interpret the call to the method `__init__(params)` on t_2, which potentially fills in the *Fields* parameter with member variables. Finally, we set *type*(`expr`) to *type*(`expr`)$\sqcup t_2$ type. CFG construction is handled in the same way as for function and method calls.
- If t is the type of an object with a defined `call` operator, i.e., is an *Instance* type, and that *Instance*, or its *Class*, contains a field called `__call__` whose type is a *Method*, then, per Python semantics, the call is treated as a call to the `__call__` method of that instance.
- For all other types, we perform no action.

If *type*(`expr`) is not a *Union*, we treat it as if it were a *union*(*type*(`expr`)).

`import` statement. These are treated as special function calls that parse and interpret the appropriate module body, return a *Module* type, and then set the type of a freshly created variable named after the module name to the returned *Module* type, unless the import statement is of the form `from a import b`. Then, a fresh variable `b` is created, and *type*(`b`) is set to *type*(`a.b`).

Exceptions `try-except-finally` (similar to `try-catch-finally` in Java), `raise`, and `with` statements. Because `try` blocks can be nested, our analysis maintains a stack of exception handlers. When analysis enters a `try` block, it first pushes onto the stack the `finally` block, if it exists, and then, from the last `except` block to the first, it pushes onto the stack a tuple consisting of that `except` block and the class types of exceptions that it handles; if a `finally` block exists, for each `except` block, an edge is added from each CFG node that exists the `except` block to the first CFG node of the `finally` block; if no `finally` block exists, then, for each `except` block, an edge is added from its last node to the first node following the last `except` block; when the analysis leaves the `try` block, the pushed entries are popped from the stack.

When analyzing a `try` block, including functions and methods called during it, from each CFG node n that may throw an exception e, if *type*(`e`) is a *Union*, then for each member t of the *Union*, we do the following:

We iterate over the stack of exception handlers from the top down, maintaining a set C of exception handlers that have possibly caught e, and a list, l, consisting of CFG nodes that are possible entry points for `finally` blocks that can be visited while handling e, initialized to the list containing only n. During the traversal, for each entry s of the stack:

- If s is an `except` handler that handles exceptions of type t_s, and t_s is compatible with t, C does not contain any exception handler that handles either t_s or its supertype, and both t and t_s are either a *Class* type with a known body or a union of such types, first add an edge from each element of l to s's entry point, then add s to C, and add all exit points of s to l. Finally, if t_s is equal to t or is its supertype, and t_s is either a *Class* type with a known body or a union of such types, stop traversing the stack.
- If s is a `finally` block, first add an edge from each element of l to s's entry point, and then set l to the list of s's exit points.

Finally, once we have iterated over the entire stack, we add an edge from each element of l to the exit point in the program.

If the type of the exception thrown is not a *Union*, we treat it as if it were a *Union* consisting of just the type of the exception thrown.

A `raise` statement that is encountered outside of a `try` block has an edge added from it to the exit point of the program.

`with` statements in Python allow for a syntactically clean way to always execute cleanup code without writing `try-finally` statements. The `with` statement looks like

```
with expression [as variable]:
    with-block
```

and Python will always execute `expression.__enter__()` when entering the `with` block, and `expression.__exit__()` when exiting it. Python will also set the value of `variable` to the value of `expression` immediately before calling `expression.__enter__()`. We convert the `with` statement to a `try - finally` statement, with appropriate local variables introduced in case `as variable` is specified.

Evals. Evals are evaluated as described in Section 4.1.1.

Definition of function, `lambda`, inner function, method, or inner method. Following Python semantics, definitions of the above are interpreted whenever they are encountered; the appropriate type is returned after interpretation; and, in all cases except `lambda`, the returned type is assigned to a freshly created variable with the name of the function or method, respectively. We interpret, in left-to-right order, the values of default parameters, and store their names and types in the $Freevars$ parameter of the type. We then build up the $Func$ type that contains the function or method name, the parameter types (usually *bot*), and the return type, which is derived by interpreting the function body assuming the *top* type for each non-default parameter, and the id of the AST node; for inner functions and methods, we add to $Freevars$ the names and types of the free variables, thus forming closures. When encountering a function definition, Python executes the expressions that are assigned to default parameters, but does not execute the function body. Thus, we generate CFG edges as usual when interpreting the values of default parameters, but we do not generate any CFG edges during the interpretation of the function body for obtaining the return type.

Definition of class. We compute a new $Class$ type where `name` and `ASTid` are derived from the class name and its root AST node, respectively; $Parents$ is derived from the inheritance specification of the class we are analyzing. We then interpret the body of the class, adding static fields and methods and their types to $Fields$ as we encounter them. Finally, we create a fresh variable named `name` in the same scope where the class definition is, and set its type to the newly computed $Class$ type.

Definition of module. We interpret its body to obtain the variables (including functions, methods, and classes) the module contains, and store their names and types in *Fields*; we set `name` to the name of the module, and set `ASTid` to the root AST node of the module. We then return the newly created *Module* type that represents the module.

Generator and generator expression. In Python, a function containing a `yield` statement is always a generator—calling the function returns a generator object that can be iterated over. We treat each generator function as a special class that inherits from a `Generator` superclass written by us, where the `next` method is derived from the generator function body. We convert any invocation of the generator into construction of the appropriate special class, for which the analysis returns the appropriate *Instance* type. Generator expressions are treated in a similar manner, and are represented as inner generator functions.

4.1.3 Alias analysis

Flow-sensitive alias analysis. We use the intraprocedural flow-sensitive alias analysis originally studied by Choi et al. [18], by extending the optimal-time algorithm for it by Goyal [44] 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 programs in general have a constant number of lexically mentioned fields.

Using types to improve alias analysis precision. We modify the algorithm to only allow alias pairs that have compatible 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 [92, 106] 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 4.2.4 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 4.2.1 and 4.2.2.

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.

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 [4] 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.

4.2 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 Duo (Q9750 at 3.8GHz) CPU with 16 GB of memory.

4.2.1 Effectiveness for optimization

Effectiveness for incrementalization. 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 4.3 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 4.3 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 4.3 summarizes our InvTS experiments. Figure 4.2 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.

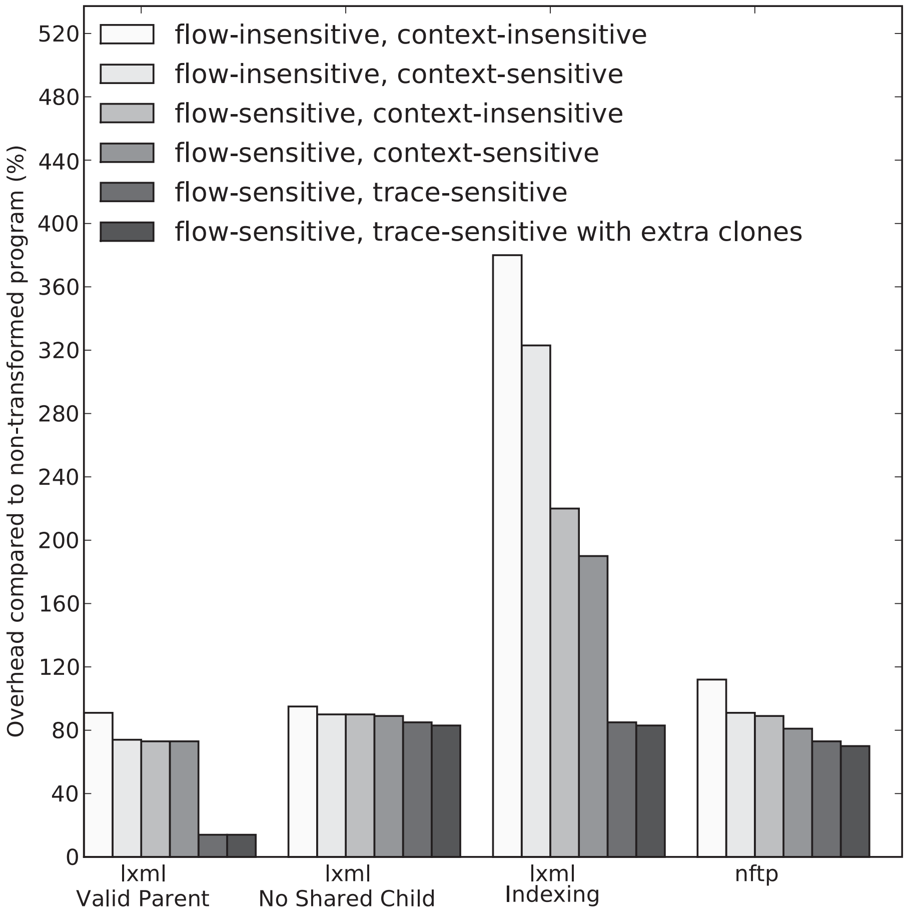


Figure 4.2: Runtime overhead of transformed programs, using precise-type-sensitive alias analysis, varying flow and context sensitivity.

Effectiveness for specialization. Psyco [84] 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 information. After collecting more information at runtime, Psyco compiles the remaining code to machine code. In our experiments, we augmented Psyco to be able to use alias information provided to it externally, enabling 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, 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 4.1 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.

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

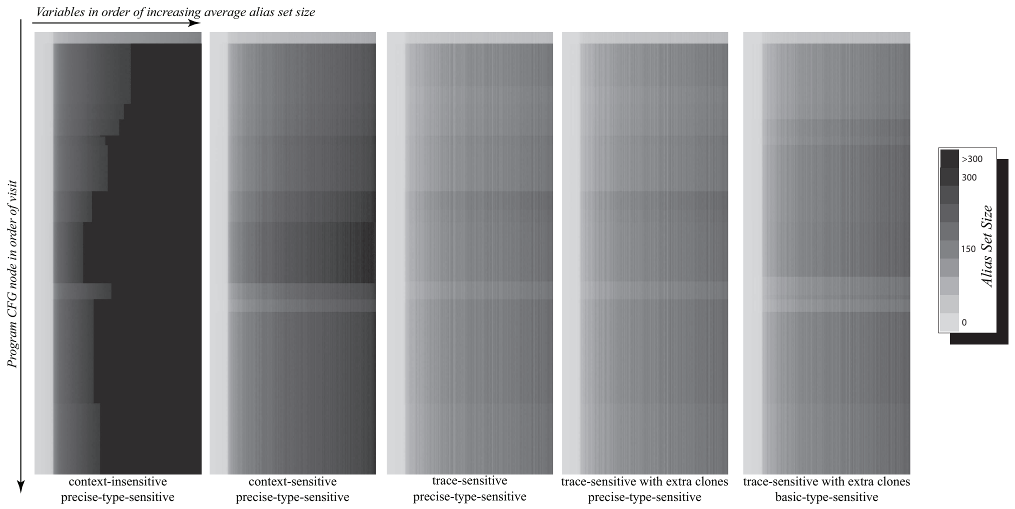


Figure 4.3: Alias set size for each variable (shown horizontally) for each CFG node (shown vertically) for flow-sensitive analysis variants for `tarfile`. Variables are ordered by increasing average alias set size in the context-insensitive precise-type-sensitive analysis.

Precision of alias analysis variants. Figure 4.3 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 4.4 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

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

Table 4.1: 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.

(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.

From Table 4.4, 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

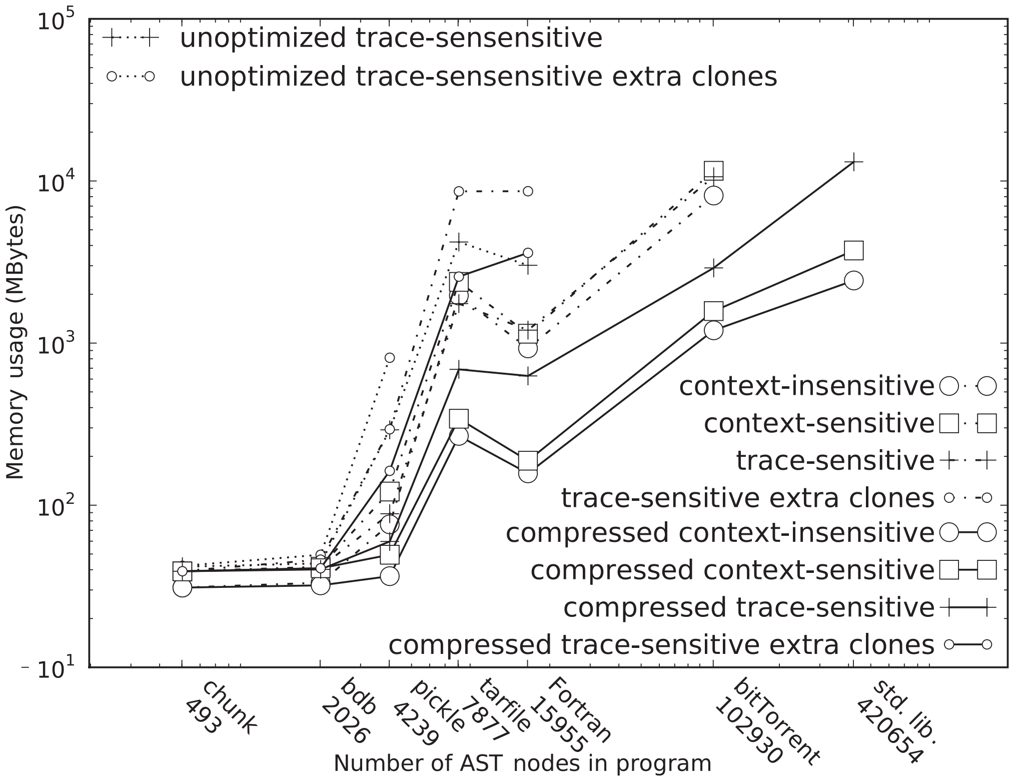


Figure 4.4: Maximum memory usage for flow- and precise-type-sensitive alias analysis variants, varying context sensitivity using uncompressed or compressed representations; "unoptimized" means that trace optimization and compression are both disabled; trace optimization is enabled for all other trace-sensitive 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.

makes it feasible for trace-sensitive analysis to analyze `bitTorrent` and `std. lib.`

Running time.

Figure 4.5 and Table 4.4 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`

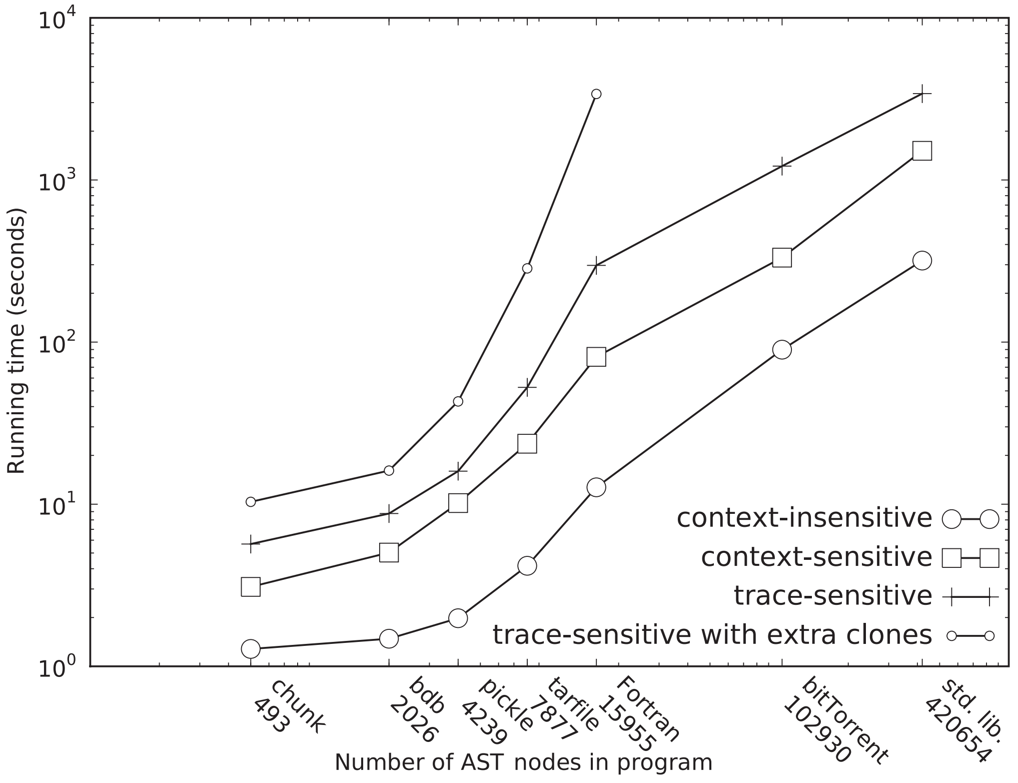


Figure 4.5: Running times for flow- and precise-type-sensitive alias analysis variants using compressed representation, varying context sensitivity. Both axes are log scale.

and `std.lib`. Without extra clones, the trace-sensitive analysis takes less than an hour to analyze `std.lib.` of 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 4.3 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 4.4 presents the data used to generate Figures 4.4 and 4.5.

4.2.3 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 4.2.1:

- We measure the effect of refinement on the precision of alias analysis

	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 4.2: 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.

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 4.2 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 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 4.6 shows that for all programs from Section 4.2.1, the program size never increases more than 11%. For programs from Section 4.2.2, 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.

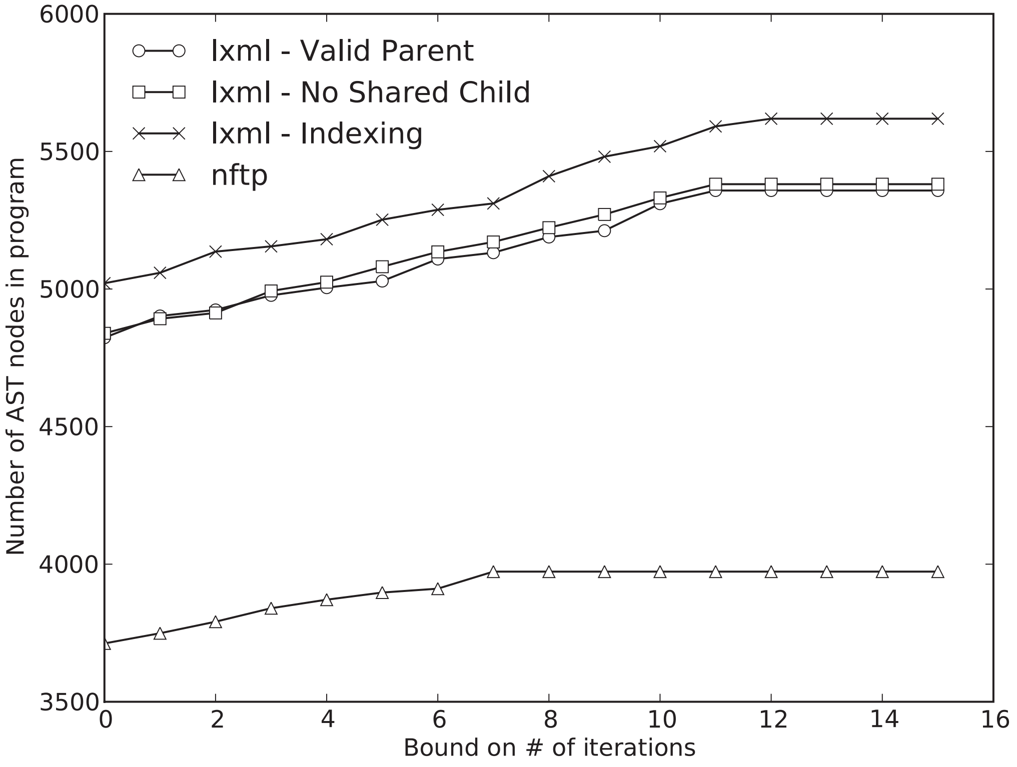


Figure 4.6: Number of AST nodes, as a function of the bound on the number of iterations of analysis and refinement.

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 4.2.1 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 4.2.1.

Figure 4.7 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.

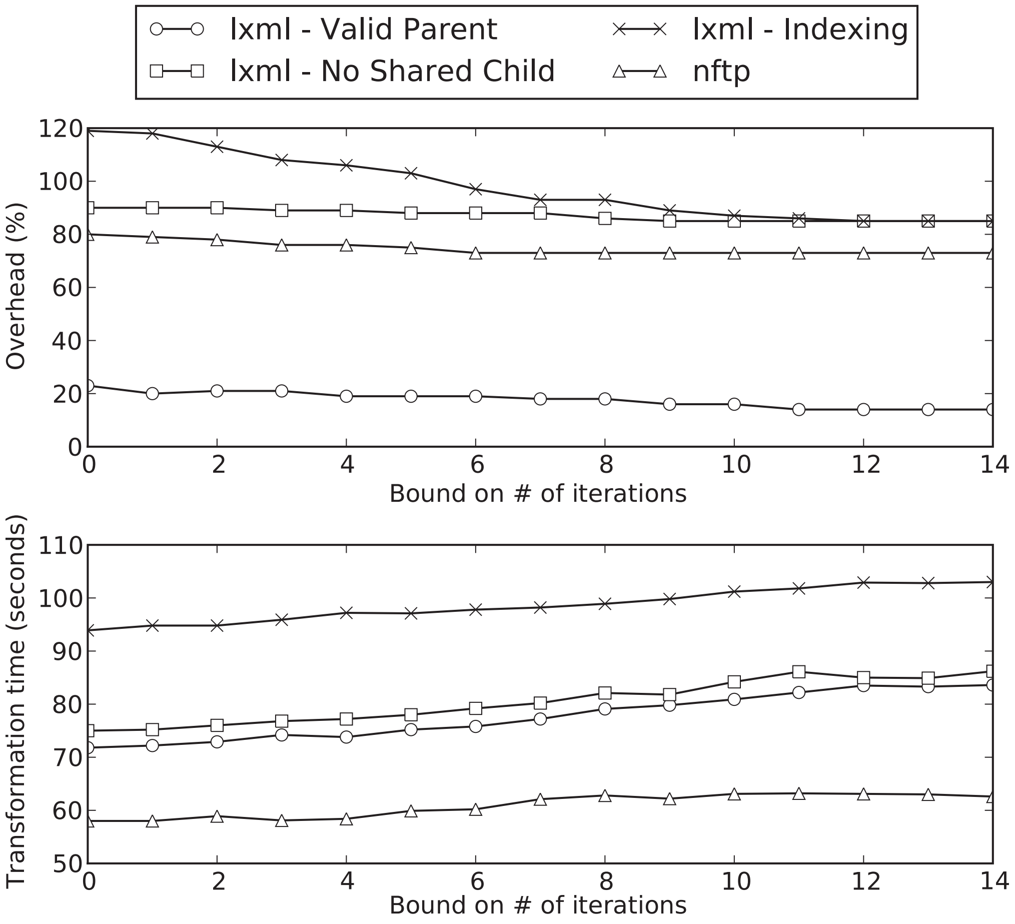


Figure 4.7: Runtime overhead of transformed programs and total program transformation time, using precise-type-sensitive alias analysis, as a function of the bound on the number of iterations of analysis and refinement.

4.2.4 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 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 4.2.1 and 4.2.2. 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.

4.3 Related work

Alias analysis and the related problem of points-to analysis have been studied extensively [50], mostly in the context of 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 [4, 99]; context-sensitive, flow-insensitive analyses, such as [35, 32, 76]; context-insensitive, flow-sensitive analyses, such as [18, 44]; and context-sensitive, flow-sensitive analyses [106, 31].

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 [51, 77] but is important in others [48]; similarly, context sensitivity provided little precision benefit in analysis of some C programs [88] but was significant for some Java applications [69].

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 [46]. 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 [63]. Their analysis is for C and is flow-insensitive, hence not appropriate for the optimizations we consider as clients.

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.

Previous work on alias analysis for dynamic object-oriented languages does not handle the breadth of dynamic features that we handle. For example, the work on alias analysis for PHP in [53, 8] does not handle first-class functions (which PHP does not support) or `eval` statements, and does not compare different variants of the analysis.

Type analysis for dynamic languages is well known to be difficult. Starkiller [89], 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 the contents of collections. The type system and type inference algorithm for a subset of JavaScript in [5] also has these limitations; in addition, it does not support field and method names as strings, functions as expressions, or `eval`. Localized Type Inference [16] for Python cannot infer types of method and procedure arguments automatically, and does not support single-value types, range types, or union types. DiamondBack [36], 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 static type analysis plays two important roles. First, the type information is used to statically determine dynamic dispatch, which is crucial to obtain a precise control flow graph [7, 98]. Second, the 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 [30], but it does not significantly help in that context, because most statements that would create such alias pairs are illegal (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 alias set 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 [62]. Unfortunately, BDDs are less effective for flow-sensitive analyses, because of the large number of strong updates to pointer information [48]. Hardekopf et al. overcome this in a partially symbolic, semi-sparse context-insensitive pointer analysis for C [48]. Adapting and evaluating those ideas in the setting of context-sensitive analysis for dynamic languages is a direction for future work.

			lxml - Valid Parent			lxml - No Shared Child			lxml - Indexing			nftp		
			97 alias checks			81 alias checks			1451 alias checks			31 alias checks		
flow sensitive	context sensitivity	type sensitivity	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 4.3: 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.

			context-insensitive						context-sensitive					
		AST	unoptimized		uncompressed		compressed		unoptimized		uncompressed		compressed	
Program	LOC	Nodes	time	memory	time	memory	time	memory	time	memory	time	memory	time	memory
chunk	172	493			1.01	31.06	1.28	31.04			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	not applicable		3.23	1964.09	4.16	267.70	not applicable		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

			trace-sensitive						trace-sensitive with extra clones					
		AST	unoptimized		uncompressed		compressed		unoptimized		uncompressed		compressed	
Program	LOC	Nodes	time	memory	time	memory	time	memory	time	memory	time	memory	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 4.4: 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.

Chapter 5

Composition of invariant rules

When improving programs, two common tasks are instrumentation and incrementalization. Instrumentation is the addition of code to monitor program behavior at runtime, to ensure that the program satisfies correctness and performance requirements. Incrementalization improves program performance by incrementally maintaining the result of an expensive computation when the values that the computation depends on are updated, rather than recomputing the result from scratch. Both improvements come at the cost of increased program complexity.

Program transformation allows us to limit this complexity, by using multiple simple, small transforms to instrument and incrementalize programs. Such simple transformations can be specified more easily as separate transformation rules, and these rules can be applied to programs one at a time to achieve the overall effects desired. Composing these simple transformation rules before applying them provides three benefits:

1. making the overall effects of the transformations clearer and easier to understand through the composed rule, without having to observe the overall effects only through the transformed subject program. Given that transformation rules are typically much smaller than subject programs, this provides a significant advantage when the rule writer desires to manually verify that the transformation rules do what he intended,

2. allowing optimizations of the composed rules, which lead to optimizations of the transformed programs that may be more difficult or impossible to realize if smaller transformation rules are applied separately. Again, this is due to the fact that transformation rules are significantly smaller than subject programs; consequently expensive — sometimes exponentially so — optimizations can be feasibly applied to them, but not to the subject programs, and

3. allowing overall faster application of transformation rules, compared with applying smaller transformations rules separately, significantly reducing the turnaround time in the edit-transform-debug cycle.

This chapter describes powerful transformations for instrumenting and incrementalizing real applications, and a method for composing transformations by composing transformation rules and optimizing the composed rules. The transformations are specified using InvTL extended for instrumentaion. The transformations are powerful because they perform sophisticated instrumentations and optimizations, especially when high-level queries over sets and objects are used. The example transformations for instrumentation are for ranking peers in BitTorrent, a peer-to-peer distributed file sharing program. The example transformations for incrementalization are for optimizing the instrumentation of BitTorrent; for efficiently computing the quality of hosts' network connections using NetFlow, a Cisco network protocol for collecting IP traffic information; and for generating efficient implementations from formal specifications for Constrained RBAC, advanced components in the ANSI standard for Role-Based Access Control (RBAC).

As in prior chapters, we specify transformations that perform incrementalization as invariant rules. Additionally, we describe minor extensions to InvTL that allow us to easily specify instrumentation rules; we then describe a method for composing transformations by composing invariant and instrumentation rules and optimizing the composed rules. We use BitTorrent to illustrate transformations for instrumentation, incrementalization, and composition. We present queries for both BitTorrent and Constrained RBAC to show that the method of composition and optimization is applicable to very different queries. We present experimental results for BitTorrent, NetFlow queries, and Constrained RBAC, and show the benefits and effectiveness of the method.

The rest of the chapter is organized as follows. Section 5.1 describes how we extended InvTL to be more suitable for specifying instrumentation rules, and introduces the running example. Sections 5.2, 5.3, and 5.4 describe instrumentation, incrementalization, and composition, respectively, for BitTorrent. Section 5.5 presents experimental results. Section 5.5.3 discusses related work.

5.1 Extending InvTL for instrumentation

We extend InvTL slightly to support instrumentation using instrumentation rules. An instrumentation rule is of the same form as an invariant rule (Chapter 1, Figure 1.2), with two exceptions:

First, the `inv` clause **inv** *result* = *computation* is replaced with the keyword `instrumentation` optionally preceded by the modifier `pure`. This indicates that the rule is for instrumentation, not for maintaining an invariant *result* = *computation*.

Second, in pure instrumentation rules, `do instead` clauses cannot be used. This means that all maintenance code must be for insertion before or after existing code, not replacement of existing code, to help ensure that pure instrumentation rules do not change program semantics.

The semantics of applying an instrumentation rule is different from applying an invariant rule in two aspects. First, for all instrumentation rules, the `do` clause under the `instrumentation` or `pure instrumentation` clause inserts code before or after the entire program, instead of before or after the computation as for an invariant rule. Second, for pure instrumentation rules, instead of automatically detecting all possible updates to the values on which a computation depends, we automatically check that all inserted code updates only new variables and fields, not existing variables and fields, using conservative static analysis first and dynamic checking for the remaining updates. This ensures that pure instrumentation rules do not change the execution of the given program except for the extra time and space for running the inserted code. This helps preserve invariants in the given program.

Finally, an instrumentation rule is applied once and only once to the given program, rather than possibly repeatedly applied as for invariant rules.

Running example. We use instrumentation and incrementalization of BitTorrent as a running example.

BitTorrent (`http://download.bittorrent.com/dl/`) is a peer-to-peer distributed file sharing protocol. When multiple peers download the same file concurrently, they can relay data to each other, making it possible for the file source to support large numbers of downloaders with only a modest increase in its load. Each peer downloads chunks of a file from other peers, and then reassembles the original file from the chunks. The set of peers that a peer communicates with is called its peer horizon.

Each chunk is sent from one peer to another as a sequence of packets, and once a chunk is completely received, the peer verifies that the chunk arrived without errors, by using an SHA1 checksum sent in a bootstrapping file that contains the checksum of each chunk of the file being distributed. If the chunk contains errors, the peer marks the sender of the chunk as untrustworthy, and attempts to retrieve the chunk from another peer.

5.2 Instrumentation of BitTorrent

Using InvTS, we instrument the BitTorrent peer to rank peers, giving lower ranks to senders and receivers with mismatches between the data packets sent and received. Doing this efficiently allows us to quickly detect bad peers or peers connected by bad links. In BitTorrent without instrumentation, such detection requires the peer to receive at least one complete chunk from another peer and thus has a delay, because checking is done at the chunk level, rather than the packet level.

An instrumented BitTorrent peer: (1) records history, i.e. sends a notification packet to all peers in its peer horizon when it receives or sends a data packet and record the notification packets received, (2) analyzes recorded history, i.e., computes the ranks of all peers in the peer horizon to reflect matches between the data packets sent and received, and (3) acts on the analysis result, i.e., prints out the list of peers in order of high to low ranks for every 1000 notification packets received. Figures 5.1 and 5.2 together show the complete instrumentation rule, explained below.

Recording history. When the BitTorrent peer receives or sends a data packet, it (1) encodes information about the sending or receiving in a byte string, containing an event type character indicating whether the peer was receiving or sending the data packet, the source and target of the data packet, and a hash of the payload of the data packet, (2) creates a notification packet containing this byte string as payload, and (3) sends the notification packet to all peers in its peer horizon. These are done by defining method `send_notification_packet` in Figure 5.1, to be called by method `send_to_horizon` in Figure 5.2, and calling `send_to_horizon` in `process` in Figure 5.2 when BitTorrent sends or receives a data packet.

When the BitTorrent peer receives a notification packet, it (1) decodes the packet into the components described above, and (2) stores the decoded information in `$sent` or `$recv` based on the event type. These are done by declaring `$sent` and `$recv` and defining method `receive_notification_packet` in Figure 5.1, and calling it in `process` when the BitTorrent peer receives a notification packet.

Analyzing recorded history. Method `compute_rank` in Figure 5.2 uses `$sent` and `$recv` to compute the rank of each peer in the peer horizon. For each peer, uniquely identified by its address *ip*, it computes *match*, the number of data packets sent by or received by the peer and whose sending and receiving match, i.e.,

$$match = |\{p : p \in \texttt{\$sent} \cap \texttt{\$recv}, p.\texttt{src} = ip \vee p.\texttt{dst} = ip\}|$$

and it computes *total*, the number of all data packets sent by or received by the peer, i.e.,

$$total = |\{p : p \in \texttt{\$sent} \cup \texttt{\$recv}, p.\texttt{src} = ip \vee p.\texttt{dst} = ip\}|$$

```
pure instrumentation

de in global py{
   import scapy  #socket module from http://www.secdev.org

   #called whenever a data packet is sent
   def send_notification_packet(peer, type, p):
     ... #send event type and info about packet p to
         #target peer using scapy over UDP on port 555

   $sent = set()  # set of all data packets sent
   $recv = set()  # set of all data packets received

   #called whenever a notification packet is received
   def receive_notification_packet(bytestring):
     ... #receive a bytestring, decode it, and insert
         #result in $sent or $recv, respectively
}
```

Figure 5.1: Instrumentation rule clauses for sending and receiving notification packets.

The peer's rank is computed as *match* divided by *total*, i.e.,

$$rank = match/total$$

Higher ranks indicate better peers. When all data packets sent by and received by a peer match, i.e., no packet is sent but not received, received but not sent, or modified in transit, the peer's rank is 1.

Acting on analysis results. For every 1000 notification packets received, we sort peers in order of high to low ranks and print out the sorted list. Method `sort_and_print` in Figure 5.2 does this.

Overhead caused by instrumentation. The overhead caused by instrumentation is the cost of sending notification packets to all peers in the peer horizon, whenever the BitTorrent peer sends or receives a data packet, and the cost of executing the queries that compute `match` and `total` for all peers in the peer horizon, whenever the BitTorrent peer receives a notification packet. This is $\mathrm{O}((S+R)^2 \times H)$ expected time, where S and R are the sizes of `$sent` and `$recv`, respectively, and H is the number of peers in the peer horizon. This is because there are a total of $\mathrm{O}(S+R)$ data packets and notification packets sent and received by each peer; there is a cost factor of $\mathrm{O}(H)$ for each packet sent or received; and computing *match* and *total* can be done in O($S + R$) expected time using hashing. The space used by the added code is O($S + R$) for storing `$sent` and `$recv`. The time spent and space used are shown on the first line in Table 5.1.

```
de in global py{
   import hashlib  #standard library
   from collections import defaultdict  #standard library
}
de in class bitTorrent py{
   #insert instrumentation at the start of method __init__
   def __init__(self):
     #start sniffing for packets sent/recv'd by current proc.
     #when a packet is sniffed, self.process is called on it
     scapy.sniff(prn = self.process)
     self.rank = defaultdict(float)  #rank for each peer
     self.packet_count = 0  #num. notif. packets received

   def process(self, packet):
     #if packet is UDP or TCP packet
     if UDP in packet or TCP in packet:
       #decode packet as a UDP or TCP packet, into p
       p = packet[UDP] if UDP in packet else packet[TCP]

       #if p's port is in the range of ports specified
       #by the BitTorrent peer, p is a data packet
       if p.port in self.portrange:
         if p.src==self.ip_addr:  #sending p
           self.send_to_horizon("s", p)
         if p.dst==self.ip_addr:  #receiving p
           self.send_to_horizon("r", p))

       #if p's port is 555, p is a notification packet
       if p.port==555:
         if p.dst==self.ip_addr:  #receiving p
           receive_notification_packet(p.payload)
           self.compute_rank()
           self.sort_and_print()
     #otherwise, we sniffed an unknown packet; do nothing

   def send_to_horizon(self, type, p):
     for peer in self.peers:
       send_notification_packet(peer, type, p)

   def compute_rank(self):
     for peer in self.peers:
       match = len(set(p for p in intersect($sent,$recv) if
               p.src==peer.ip_addr or p.dst==peer.ip_addr))
       total = len(set(p for p in union($sent,$recv) if
               p.src==peer.ip_addr or p.dst==peer.ip_addr))
     self.rank[peer] = 1.0 if total==0 else 1.0*match/total

   def sort_and_print(self):
     self.packet_count += 1
     if self.packet_count % 1000 == 0:
       for (peer,rank) in self.rank.items().sorted(
                    lambda rank1,rank2: rank1[1]-rank2[1]):
         print "peer: ", peer.ip_addr, "rank: ", rank
}
```

Figure 5.2: Clauses to instrument BitTorrent peer to process packets received, compute ranks, and print sorted peers.

5.3 Decomposition and incrementalization

Non-trivial queries are usually expensive. When such queries need to be performed repeatedly and the values they depend on change constantly, efficient computations of the queries rely on maintaining the results of the queries incrementally as the values depended on change.

In the instrumentation for BitTorrent, the queries for computing `match` and `total` are expensive. They take expected time proportional to the size of `$sent` plus the size of `$recv`, and this cost is incurred for each notification packet received. To make the queries efficient, we maintain their results incrementally as data packets are sent and received, i.e. as `$sent` and `$recv` are updated. We do this for each peer in the peer horizon.

We could use a previously studied method [73, 71] to incrementalize expensive queries. It incrementalizes each query in a basic form using a set of coordinated transformations specified in an invariant rule; the transformations replace the query with a retrieval of the query result from a variable, and insert code to maintain the query result at all places that update the values the query depends on. For nested queries, the effect is that the innermost query in a basic form is incrementalized first; after this the query is replaced by a retrieval of its result from a variable, and then the outer query is in a basic form and is incrementalized next; this continues until the outermost query is incrementalized.

This previous method of repeatedly applying invariant rules has three drawbacks: (1) the overall result of incrementalizing a nested query is difficult to understand because it is scattered in many places in the final transformed program, (2) optimizations enabled by incrementalization are hard to perform on the often large and complex transformed program, and (3) repeatedly applying invariant rules is expensive because update analysis and other analyses of the entire program are required before applying each rule.

To overcome these drawbacks of the previous method, we will:

1. decompose nested queries into subqueries in basic forms,

2. derive invariant rules for the subqueries using the previously studied method

3. compose and optimize rules derived in step 2 - described in Section 5.4.

We will then use InvTS to apply the composed and optimized rules obtained in step 3.

We describe below the decomposition into subqueries and the derivation of invariant rules for subqueries. We describe composition and optimization in the next section.

Decomposing nested queries. The parameters of a query are the variables used in the query excluding variables introduced in the query. To decompose nested queries, we do three steps. Step 1 follows the dependency order of computation, extracts subqueries in innermost, leftmost-first order. That is, if a subquery is contained inside another subquery, then the inner one is extracted first; if neither of two subqueries is contained inside the other, then the left one is extracted first. Then, step 2, for each subquery, introduces a map from tuples of values of the subquery parameters to subquery results, and finally step 3 rewrites the original query to use this map in place of the subquery.

For the query for computing `match` in Figure 5.2, step 1 extracts from

```
len(set(p for p in intersect($sent,$recv)
        if p.src==peer.ip_addr or p.dst==peer.ip_addr))
```

the inner subquery `intersect($sent,$recv)`, step 2 introduces a new map `$I` to store the query's results, and step 3 replaces `intersect($sent,$recv)` in the outer query with `I[($sent,$recv)]`. We thus obtain

```
$I[($sent,$recv)]     = intersect($sent,$recv)
match                 = len(set(p for p in $I[($sent,$recv)]
                              if (p.src==$ip or p.dst==$ip))
```

Repeating this procedure until we reach the outermost query, we end up introducing maps `$I`, `$P`, and `$M` storing the intersection, the selected set for each peer, and the results of the entire query for `match`, respectively; we also replace the original query by`$M[($sent,$recv,peer.ip_addr)]`.

```
$I[($sent,$recv)]     = intersect($sent,$recv)
$P[($sent,$recv,$ip)] = set(p for p in $I[($sent,$recv)]
                              if (p.src==$ip or p.dst==$ip))
$M[($sent,$recv,$ip)] = len($P[($sent,$recv,$ip)])
```

Parameters that throughout the lifetime of the program are bound to just one object — `$sent` and `$recv` in this example — are unnecessary and thus removed, yielding the following forms of subqueries. The original query for computing `match` is then replaced by `$M[peer.ip_addr]`.

```
$I      = intersect($sent,$recv)
$P[$ip] = set(p for p in $I if p.src==$ip or p.dst==$ip)
$M[$ip] = len($P[$ip])
```

For the query for computing `total`, we do the same, yielding three subqueries also, one for the union, one for the selection, and one for the result.

instrumented BitTorrent variant	time	space
use no inv. rules	$O((S+R)^2 \times H)$	$S+R$
use separate inv. rules	$O((S+R) \times H)$	$5(S+R)$
use composed inv. rules	$O((S+R) \times H)$	$5(S+R)$
use opt. composed inv. rules	$O((S+R) \times H)$	$S+R$

Table 5.1: Time and space overhead caused by instrumentation. S and R are the sizes of `$sent` and `$recv`, respectively. H is the maximal number of peers in the horizon of any given peer.

Deriving invariant rules for subqueries. We derive invariant rules for each query for each kind of update to a parameter of the query using a previously studied method [73, 71, 87]. The method works for a large class of queries and updates. Figure 5.3 shows the resulting invariant rules for incrementalizing `$I`, `$P`, and `$M`, respectively. Rules for incrementalizing the query for `total` are similar.

Overhead caused by instrumentation using separate invariant rules. The overhead caused by instrumentation after incrementalization using separate invariant rules is shown on the second line in Table 5.1. The time complexity is $O((S + R) \times H)$, because each piece of maintenance code inserted takes constant time, and retrieving query results from all three maps also takes constant time, and thus the overhead is constant for each peer in the peer horizon for each packet sent or received. The space complexity is bounded by $(S + R) \times 5$ because, besides storing `$sent` and `$recv`, we also store `I` and `P` for computing the query for `match` and two similar variables for computing the query for `total`, and the space for each of these four maps is bounded by $S + R$; the result map `M` for the query for `match` and the result map for the query for `total` take significantly less space and thus are omitted.

5.4 Composition

This section presents how we perform:

1. composition of rules applicable to the subqueries obtained from decomposing a nested query, and

2. optimization of composed invariant rules from step 1.

```
inv py{ $I } = py{
   intersect($sent,$recv)
}
de in class bitTorrent py{
   def __init__(self):
     $I = set()
}
at py{ $sent.add($p) }
do before py{
   if $p in $recv:
     if $p not in $I:
       $I.add($p)
}
at py{ $recv.add($p) }
do before py{
   if $p in $sent:
     if $p not in $I:
       $I.add($p)
}
```

```
inv py{ $P[$ip] } = py{
   set(p for p in $I if
     p.src==$ip or p.dst==$ip)
}
de in class bitTorrent py{
   def __init__(self):
     $P = defaultdict(set)
}
at py{ $I.add($p) }
do before py{
   if $p not in $P[$p.src]:
     $P[$p.src].add($p)
   if $p not in $P[$p.dst]:
     $P[$p.dst].add($p)
}
at py{ $I.remove($p) }
do before py{
   if $p not in $P[$p.src]:
     $P[$p.src].remove($p)
   if $p not in $P[$p.dst]:
     $P[$p.dst].remove($p)
}
```

```
inv py{ $M[$ip] } = py{
   len($P[$ip])
}
de in class bitTorrent py{
   def __init__(self):
     $M = defaultdict(int)
}
at py{ $P[$ip].add($p) }
do before py{
   $M[$ip] += 1
}
at py{ $P[$ip].remove($p) }
do before py{
   $M[$ip] -= 1
}
```

Figure 5.3: Invariant rules for maintaining the results of subqueries in `$I`, `$P`, and `$M`. Clauses for handling removals from `$sent` and `$recv` are symmetric to clauses for handling addition and are omitted for brevity.

5.4.1 Composition of rules

The transformations we use for composition respect language semantics so that applying the composed rule yields programs that have the same semantics as programs obtained by applying individual rules, except that the former programs can be more efficient due to optimizations performed during composition.

As described in Section 5.3, a nested query q gets decomposed into a sequence of invariants $r_1 = q_1, r_2 = q_2, \ldots, r_n = q_n$, where the result r_n of the last invariant, with its parameters instantiated by an appropriate substitution σ, equals the original query q. For example, for the incrementalization of `match`, the query is decomposed into three invariants, and the value of the original query is obtained by instantiating the parameter `$ip` of the result `$M[$ip]` of the third invariant with `peer.ip_addr`. For each of these invariants $r_i = q_i$, there is a corresponding invariant rule R_i of the form `inv` $r_i = q_i \;\; B_i$, where B_i is the body of the rule. For simplicity, we assume that B_i does not contain `if` clauses; otherwise static analysis may be used to evaluate or simplify the `if` clauses in applying the transformations. Composition produces a single rule whose invariant is $r_n = q'_n$ such that q'_n instantiated with σ is syntactically identical to the original query q.

Our composition algorithm builds this rule up starting from the first rule, which is for the innermost subquery of the original query. The algorithm succeeds for rules that obey the following: (1) every update introduced by the maintenance code of the rule for an inner query is handled by the update pattern of the rule for the enclosing query, (2) every update pattern in the rule for an outer query updates the query result from the rule for the enclosed query, and (3) the maintenance code in a rule does not create aliases to fresh variables introduced by the rule. The construction produces a sequence of rules $R'_1, R'_2, \ldots, R'_n$, where R'_i is the result of composing the first i invariant rules, namely, R_1 to R_i. As the base case, R'_1 is identical to R_1. At the end, R'_n is the desired rule characterized above.

The algorithm is as follows, where $t_1[v \mapsto t_2]$ denotes t_1 with each occurrence of v replaced with t_2.

```
$q'_1 = q_1; B'_1 = B_1; R'_1 = R_1$                              (1)
for $i = 1$ to $n - 1$                                            (2)
    $q'_{i+1} = q_{i+1}[r_i \mapsto q'_i]$                        (3)
    $B'_{i+1} = \texttt{transform}(B'_i, B_{i+1})$                (4)
    $R'_{i+1} = \textbf{inv}\ r_{i+1} = q'_{i+1}\ B'_{i+1}$       (5)
```

Intuitively, applying the substitution $[r_i \mapsto q'_i]$ to q_{i+1} in line (3) reconstructs part of the structure of the original nested query, because this substitution is the "opposite" of the replacement of q_i with r_i performed during decomposition of the query. This substitution is valid because R'_i ensures its invariant

$r_i = q'_i$. To ensure that R'_{i+1} also maintains this invariant, the body B'_i of R'_i is used in line (4) as the basis for the body B'_{i+1} of R'_{i+1}. To ensure that R'_{i+1} also maintains the invariant of R_{i+1}, the transformation specified by B_{i+1} is applied to the maintenance code in B'_i. Specifically, `transform`(B'_i, B_{i+1}) in line (4) returns the result of that application. Following the semantics for rules in Section 5.1, `transform` first checks whether every update to parameters of q_{i+1} in B'_i matches at least one *update* pattern in B_{i+1}. If so, declarations and maintenance code in B_{i+1} are inserted in B'_i as specified by the **de** and **do** clauses in B_{i+1}. If not, `transform` aborts; this causes the composition algorithm to abort.

Note that the transformation defined by B_{i+1} is applied only to code in B'_i. Normally—i.e., if we did not use rule composition—it would be applied to the entire subject program. To ensure that applying it only to B'_i gives the same result as applying it to the entire subject program, `transform`(B'_i, B_{i+1}) checks that every update pattern in B_{i+1} updates the query result r_i introduced by R'_i and hence does not match any update in the subject program. If this condition is not satisfied, `transform` aborts.

`transform` needs may-alias information to identify possible updates to query parameters. `transform`(B'_i, B_{i+1}) checks whether fresh variables introduced by B'_i, namely, variables represented by metavariables that do not appear in the query in the `inv` clause, may become aliased by assignments in B'_{i+1}. If so, the call to `transform` aborts; otherwise, it proceeds knowing that those variables have no aliases. `transform` makes no assumptions about possible aliases of other variables.

Figure 5.4 shows the composition of the three rules in Figure 5.3.

5.4.2 Optimization of composed invariant rules

Performing optimizations on the maintenance code in invariant rules (before applying the rules) conveniently allows the invariants associated with the rules to be exploited during optimization. While these invariants could, in principle, be made available to an optimizer running on the transformed program, it is more efficient to optimize the rules than the transformed program, which is typically much larger. Optimization of maintenance code is especially useful for rules generated by composition, because composition may introduce redundant or unreachable computations.

We introduce three optimizations: (1) membership test simplification, (2) dead branch elimination, and (3) dead variable elimination. We repeatedly apply these optimizations to the rules until none are applicable.

(1) Membership test simplification. The combination of decomposing nested queries, incrementalizing them, and finally composing them produces,

```
at py{ $sent.add($p) }
do before py{
   if $p in $recv:
     if $p not in $I:
       if $p not in $P[$p.src]:
         $M[$p.src] += 1
         $P[$p.src].add($p)
       if $p not in $P[$p.dst]:
         $M[$p.dst] += 1
         $P[$p.dst].add($p)
       $I.add($p)
}
```

```
at py{ $recv.add($p) }
do before py{
   if $p in $sent:
     if $p not in $I:
       if $p not in $P[$p.src]:
         $M[$p.src] += 1
         $P[$p.src].add($p)
       if $p not in $P[$p.dst]:
         $M[$p.dst] += 1
         $P[$p.dst].add($p)
       $I.add($p)
}
```

```
at py{ $sent.remove($p) }
do before py{
   if $p in $recv:
     if $p in $I:
       if $p in $P[$p.src]:
         $M[$p.src] -= 1
         $P[$p.src].remove($p)
       if $p in $P[$p.dst]:
         $M[$p.dst] -= 1
         $P[$p.dst].remove($p)
       $I.remove($p)
}
```

```
at py{ $recv.remove($p) }
do before py{
   if $p in $sent:
     if $p in $I:
       if $p in $P[$p.src]:
         $M[$p.src] -= 1
         $P[$p.src].remove($p)
       if $p in $P[$p.dst]:
         $M[$p.dst] -= 1
         $P[$p.dst].remove($p)
       $I.remove($p)
}
```

Figure 5.4: Result of composing the rules for incrementalization of `match` in Figure 5.3. The `inv` clauses and `de` clauses are not shown; they are the same as in the optimized rule for incrementalizing `match` on the left in Figure 5.5, except that, in the `de` clause, the definition of `__init__` also contains the initializations `$P = defaultdict(set)` and `$I = set()`.

in the composed maintenance code, membership tests that are redundant, i.e., membership tests that can be statically evaluated to `true` or `false`. We illustrate how to statically evaluate (in two steps) a membership test by doing so to code from the third column of Figure 5.4:

```
if $p in $I:
  if $p in $P[$p.dst]:
    $M[$p.dst] -= 1
```

In step **i**, using the invariant, derived during decomposition, that:

```
$P[$ip] = set(p for p in $I if p.src==$ip or p.dst==$ip)
```

we rewrite the above to:

```
if $p in $I:
  if $p in set(p for p in $I if p.src==$p.dst or p.dst==$p.dst):
    $M[$p.dst] -= 1
```

which is equivalent to:

```
if $p in $I:
  if $p in $I and ($p.src==$p.dst or $p.dst==$p.dst):
    $M[$p.dst] -= 1
```

In step **ii**, we simplify the above using simplification of Boolean-valued expressions in context:

```
if $p in $I:
  if true and ($p.src==$p.dst or $p.dst==$p.dst):
    $M[$p.dst] -= 1
```

and then, using simplification of equality, to:

```
if $p in $I:
  if true and ($p.src==$p.dst or true):
    $M[$p.dst] -= 1
```

and again, using simplification of Boolean expressions, to:

```
if $p in $I:
  if true:
    $M[$p.dst] -= 1
```

In general, membership test simplification simplifies, in two steps, membership tests of the form v `in` r such that $r = q$ is an invariant generated during query decomposition and q has the form `set(`x `for` x `in` S `if` c`)`.

In step **i** we replace a membership test v `in` r with the equivalent v `in` S `and` $c[x \mapsto v]$, where $c[x \mapsto v]$ denotes c with all occurrences of x replaced with v.

In step **ii**, we simplify the conjuncts from step **i** by repeatedly applying, until we reach a fixed-point, (a) simplification of Boolean-valued expressions in context, (b) simplification of equality, and (c) simplification of Boolean expressions.

ii(a) — simplification of Boolean-valued expression in context — if any conjunct simplifies to either an expression that appears as the condition of an enclosing `if` statement or the negation of such an expression, and if the variables on which the conjunct depends are not updated before the membership test in the branch in which it appears (checking this relies on alias analysis, which is handled the same way as when transforming maintenance code), then the conjunct is replaced with `true` or `false`, as appropriate;

ii(b) — simplification of equality — if two expressions without side effects are compared, and they are the same expression, then the comparison is replaced with `true` (e.g., `x==x` simplifies to `true`);

ii(c) — simplification of Boolean expressions — standard Boolean simplifications are applied to the resulting conjunction (e.g., *cond* `and true` simplifies to *cond*).

If either **ii**(a), **ii**(b), or **ii**(c) replaces any conjunct with a Boolean constant, then the original membership test is replaced with the result of simplifying the conjunction; otherwise, the membership test is left unchanged.

It would be difficult to perform this optimization based purely on analysis of the transformed program because step **i** would require re-discovering the invariant from the invariant rule.

(2) Dead branch elimination. If the condition in an `if` statement is a boolean constant (typically as a result of membership test simplification), then the entire `if` statement is replaced with the reachable branch. Continuing the above example, this optimization replaces

```
if true:
  $M[$p.dst] -= 1
```

with

```
$M[$p.dst] -= 1
```

(3) Dead variable elimination. If the value of a variable that is introduced by a transformation rule is not used in the rule's result (on the left side of its `inv` clause) or in the rule's maintenance code (in its `do` clauses), and there are no aliases of the variable, then the variable and all updates to it can be eliminated. For example, after repeatedly applying membership test simplification to the rules in Figure 5.4, these conditions hold for `$I` and `$P`, so these variables and updates to them are eliminated.

Applying these optimizations to the composed rule for `match` in Figure 5.4 and the similar composed rule for `total`, we obtain the optimized composed rules in Figure 5.5. Note how much easier the `at` clauses in these rules are to understand than those in Figure 5.4.

Overhead caused by instrumentation using optimized composed rules. The optimized composed invariant rule for computing `match` does not introduce `$I` and `$P`. Similarly, the optimized composed invariant rule for computing `total` does not introduce maps maintaining the union and peer selection. Thus, the optimizations eliminate four maps, each of size $O(S + R)$. This is reflected in the improved space complexity in the last line in Table 5.1.

5.4.3 Composing instrumentation rules with invariant rules

Our system composes instrumentation rules with invariant rules by applying invariant rules, including composed invariant rules, to the code in instrumentation rules, before applying the instrumentation rules to a subject program. This allows expensive queries in instrumentation code to be incrementalized before the instrumentation code is inserted in a subject program. When applying an invariant rule to the code in an instrumentation rule, alias analysis, etc., are done in the same way as when composing invariant rules.

This technique is not essential, but it reduces the overall transformation time, for the following reason. Before applying a transformation rule, InvTS analyzes the code being transformed. Thus, applying an invariant rule to an instrumentation rule and applying the resulting rule to the subject program, instead of sequentially applying the instrumentation rule and then the invariant rule to the subject program, trades two analyses of the subject program for one analysis of the code in the instrumentation rule and one analysis of the subject program. This increases performance, because the code in an instrumentation rule is typically much smaller than the subject program, and because the alias analysis used to analyze code in instrumentation rules is less sophisticated, and hence cheaper, than the alias analysis used to analyze subject programs.

```
inv py{ $M[$ip] } = py{
    len(set(p for p in intersect($sent,$recv)
              if p.src==$ip or p.dst==$ip))
}
de in class bitTorrent py{
   def __init__(self):
     $M = defaultdict(int)
}
at py{ $sent.add($p) }
do before py{
   if $p in $recv:
     if not ($p in $sent):
       $M[$p.src] += 1
       $M[$p.dst] += 1
}
at py{ $recv.add($p) }
do before py{
   if $p in $sent:
     if not ($p in $recv):
       $M[$p.src] += 1
       $M[$p.dst] += 1
}
```

```
inv py{ $T[$ip] } = py{
    len(set(p for p in union($sent,$recv)
              if p.src==$ip or p.dst==$ip))
}
de in class bitTorrent py{
   def __init__(self):
     $T = defaultdict(int)
}
at py{ $sent.add($p) }
do before py{
   if $p not in $recv:
     if not ($p in $sent):
       $T[$p.src] += 1
       $T[$p.dst] += 1
}
at py{ $recv.add($p) }
do before py{
   if $p not in $sent:
     if not ($p in $recv):
       $T[$p.src] += 1
       $T[$p.dst] += 1
}
```

Figure 5.5: Optimized composed rules for maintaining `match` and `total`.

5.5 Experiments

To confirm that composing and optimizing invariant rules and instrumentation rules can increase the efficiency of the transformed program and reduce transformation time, as predicted, we performed experiments on three applications: BitTorrent, a NetFlow query tool, and Constrained RBAC. We developed an automated tool that transformed each application using three transformation variants:

- Application of separate rules, in dependency order.
- Composition of rules, followed by application of the composed rule.
- Composition and optimization of rules, followed by application of the optimized rule.

For each variant, we measured the size of the application before and after the transformation, the time it took to compose the rules, the time it took to optimize the composed rules, the time it took InvTS to apply the rules, and quantities about the transformed programs. All programs were written in Python and run using Python 2.6.1.

We also perform experiments that show the correctness and performance of Constrained RBAC incrementalized in several ways.

5.5.1 BitTorrent

We wrote rules that modify the BitTorrent peer to send notifications to all peers in its peer horizon and to receive and efficiently analyze notifications sent by its peers. The analysis determines and reports, for each peer in the peer's horizon, how good that peer's connection is. When the six rules totaling 171 lines are separately applied to the BitTorrent peer, the BitTorrent peer increases from 41,162 to 41,374 lines of code, a difference of 212 lines.

Optimization of composed rules. To measure the effect of different methods of rule composition on the CPU and memory overhead of the instrumentation, we performed experiments on BitTorrent peers transformed by (1) separate rule application, (2) composed rule application, and (3) optimized composed rule application. In each experiment, we measure the number of notifications stored by the instrumentation, the number of set operations performed by the instrumentation, and the CPU and network usage.

During each experiment, we transferred a 1GB file from a BitTorrent peer to 29 other BitTorrent peers over a 100 MBit link. Each peer was on a virtual machine running Ubuntu 9.04 with 1GB of RAM and a single core of a Xeon

L5430 `@` 2.66GHz provisioned to it. As the peers were never CPU-bound, CPU under-provisioning was not an issue.

Table 5.2 presents the experimental results. Without optimization, the BitTorrent peers store 93 million notifications, and perform 190 million additional set operations. Optimizing the composed rules eliminates storage of intermediate query results and thereby eliminates about two thirds of this overhead, reducing the number of stored notifications to 25 million, and the number of additional set operations to 59 million.

The CPU usage of the original BitTorrent peer and the BitTorrent peer obtained by optimized composed rule application are nearly indistinguishable. The additional 7% of CPU use by the other two variants is due to the maintenance of intermediate query results. As none of the BitTorrent variants were CPU bound, the additional CPU usage did not affect the total time it took for the file to be transferred to 29 peers, which was approximately 220 seconds.

Rule application time. Table 5.2 shows that separate rule application takes the longest time: 2,998 seconds. Composed rule application takes 2,320 seconds, after taking less than 3 seconds to compose the rules, a net savings of 675 seconds. Optimizing the composed rule takes under 4 seconds, and reduces rule application time to 2,261 seconds, a further gain of 55 seconds.

Effects of instrumentation on free-riding clients. There are non-specification-adhering modifications to BitTorrent clients that attempt to get around the BitTorrent choking feature that prevents specification-adhering clients from sending data to free-riding clients [78]. One such modification has the peer start sending out chunks of the torrent before the peer has fully downloaded them. This self-promotion causes no harm when there are few or no network errors, but it makes the swarm susceptible to swarm poisoning — wide propagation of chunks corrupted by network errors — when network errors increase. To measure the effect of swarm poisoning, we transferred a 1GB file from a BitTorrent peer to 29 other BitTorrent peers over a 100 MBit link, with 3 of 29 peers having a 10% error rate. This took 438 seconds and a total bandwidth of 93.1GB. This is over 2 times as long, and a factor of 3 increase in total bandwidth used, compared to the specification-adhering BitTorrent swarm. Note that this is 10% error rate in 10% of the peers, so only 1% overall error rate.

To combat swarm poisoning, we modify our BitTorrent instrumentation rule to use the computed ranking to let the peer avoid connecting to peers with low ranks. We do so by modifying the BitTorrent metric for selecting peers, stored in field `goodness` of each peer, to prefer peers with better ranks. This is a change to the existing code in BitTorrent, so we remove `pure` from `pure instrumentation`. We measure the effect of this instrumentation by performing the same experiment as above, but using the clients transformed using the impure instrumentation rule. The experiment shows that the swarm took 227 seconds and a total bandwidth of 34.2GB to transfer the same 1GB file over a 100 MBit link, which is comparable to the performance of a specification-adhering swarm.

5.5.2 NetFlow

NetFlow is an IETF-standardized [20] network protocol used for analyzing network traffic. In NetFlow, source hosts collect information about their network activity, including information about packets received and sent. They then transmit this information using the NetFlow protocol to a target host, called a NetFlow collector. The collector may analyze the received information on-the-fly, store it for further analysis, or discard it if it cannot cope with the volume of the incoming information.

We created a NetFlow query tool based on the collector from the `flow-tools` package [86]. Pseudocode for this tool is given in Figure 5.6. This tool allows the execution of a user-specified query function over the sets `SENT`, `RECV`, and `HOSTS` — the set of packets sent by the hosts, the set of packets received by the hosts, and the set of hosts, respectively. A NetFlow query is a Python function that is executed every time a packet is received.

Queries can be written easily and implemented efficient using our NetFlow query tool. Figure 5.7 shows an example query similar to the query for BitTorrent instrumentation for ease of explanation. It computes, for each host, the quality of its network connection, defined as the fraction of packets sent to or received by the host that arrive unchanged. This is computed as `match/total`, where `match` is the number of packets that were sent to the host, received by the host, and not modified in transit, and `total` is the total number of packets sent to the host, including packets that were lost or changed.

To determine the effect of incrementalization on this NetFlow query, we ran the non-incrementalized and the variously incrementalized variants of the query on a set of 10 million packets recorded over the course of approx. 20 seconds from a saturated Gigabit network with 5 hosts on it. We set the time limit for the query program to 600 seconds. The query was run on an Intel $i7$ 920@3.1GHz with 12GB of RAM, running Ubuntu 9.04.

Incrementalization and rule composition. As shown in Table 5.2, "Original query" row, the query program generated from the NetFlow query from Figure 5.7 exceeds the time limit of 600 seconds while processing an average of only 81 packets per second. This is because the `total` and `match` set comprehensions iterate over the entire `SENT` and `RECV` sets every time the NetFlow query is called.

```
HOSTS = set()
RECV = set()
SENT = set()

for p in generate_netflow_packets():
    if p.is_received:
        RECV.add(p)
    else:
        SENT.add(p)

    HOSTS.add(p.dst)

    query()
```

Figure 5.6: Pseudocode for the NetFlow query tool.

```
def query():
  for host in HOSTS:
    match = len(set(p for p in intersect(SENT,RECV)
                      if p.dst==host))
    total = len(set(p for p in union(SENT,RECV)
                      if p.dst==host))
    quality[host] = 1.0*match/total
```

Figure 5.7: The NetFlow query function.

It is clear that for reasonable performance, the results of both set comprehensions should be incrementally maintained. We do so by applying five invariant rules to the query program. These rules are very similar to the rules in Figure 5.3, the rules used to incrementalize the instrumentation of BitTorrent. As with BitTorrent, we transformed the query program by separate rule application, by composed rule application, and by optimized composed rule application. For each transformed query program, we measured how long it takes to analyze 10 million packets.

The last three rows of Table 5.2 present the results of the experiments. The query programs transformed by separate application or composed application took approximately 33 seconds to process 10 million packets. The query program transformed by optimized composed application took 19.9 seconds to process the same data. As the packets were recorded over the course of 20 seconds and there is non-negligible overhead in reading the packets from disk, we can infer that the query program transformed by optimized composed application is capable of running the query in real-time without the need to store the packets to disk. These experiments show that application of optimized and composed invariant rules provides very tangible benefits over separate application of rules.

Rule application time. Table 5.2 shows that separate application of rules takes the longest time: 21 seconds. Applying the composed and the optimized composed rules takes 15 seconds each, with composition taking an additional 1 second, and optimization taking another 0.5 seconds.

5.5.3 Constrained RBAC

RBAC is an ANSI-standardized [3] framework for controlling user access to resources based on roles. It can significantly reduce the cost of security policy administration and is increasingly widely used in large organizations. Core RBAC controls access based on relations between permissions, users, sessions, and roles. Core RBAC and its incrementalization is described in Chapter 3. Constrained RBAC adds to Core RBAC:

- Static separation of duty (SSD) constraints — a user can be assigned to at most `c` roles from `rs`, a set of roles.
- Dynamic separation of duty (DSD) constraints — a session can have at most `c` roles from a set `rs` of roles active at the same time.

All commands are inherited from `CoreRBAC` except that `AssignUser` is redefined to also check that the current state of Constrained RBAC satisfies the SSD and DSD constraints. New administrative commands, `Create-/DeleteSsd/DsdSet`, `Add/DeleteSsd/DsdRoleMember`, and `SetSsd/DsdSet-Cardinality` are added to create, modify, and delete SSD and DSD constraints; the non-deletion commands check that the new or updated SSD and DSD constraint would be satisfied, and that the cardinality would be in the required range. New review functions, `Ssd/DsdRoleSets`, `Ssd/DsdRoleSet-Roles`, and `Ssd/DsdRoleSetCardinality` are introduced to query SSD/DSD constraints. Supporting system functions and advanced review functions are simply inherited.

The SSD constraint, defined in method `CheckSSD()`, says that assignment of roles to users must satisfy:

```
forall u in USERS, [name,c] in SsdNC |
   #{r: r in AssignedRoles(u) | [name,r] in SsdNR } <= c
```

Mirroring the Z specification, we extended the 125-line straightforward implementation of Core RBAC [71] into a 381 line straightforward implementation of Constrained RBAC. Unfortunately, such a specification is grossly inefficient when evaluating `CheckAccess`, the main query of RBAC.

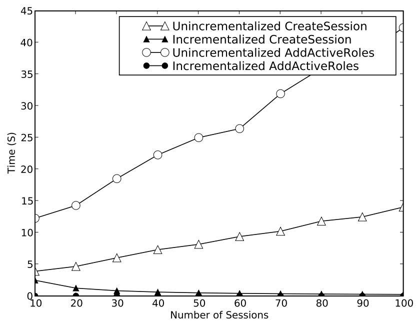
To rectify this, we applied 21 invariant rules to the straightforward implementation, incrementalizing all queries in it. Out of the 21 rules, only 7 are unique to Constrained RBAC; the other 14 are the same as the rules used to incrementalize Core RBAC. When the straightforward Constrained RBAC program is incrementalized, it becomes 2183 lines of code, a more than 5-fold increase in size. In contrast, when Core RBAC was incrementalized in [71], it tripled in size to a bit over 400 lines. Incrementalization of queries speeds them up asymptotically: for example, `CheckAccess` goes from $O(roles)$ to $O(1)$, which in our experiments with 100 roles manifests itself as an almost 50-fold speedup.

Rule application time. We verify that it is faster to apply composed invariant rules than to separately apply non-composed invariant rules by measuring how long it took to incrementalize Constrained RBAC using both methods. No dead code is eliminated by the optimizations, so the optimized composed rules are identical to the composed rules.

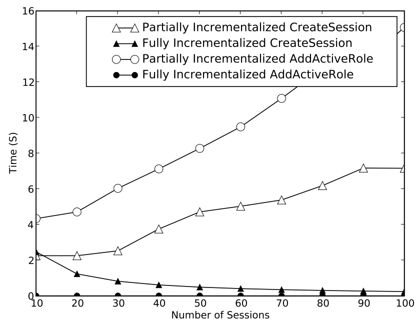
Table 5.2 shows that applying the composed rules is much faster than separately applying the non-composed rules, taking 44 rather than 257 seconds. The reason is evident from the "# `inv` clauses applied" column, which shows that when InvTS applies the composed rules, it tries to apply fewer `inv` clauses than when applying the non-composed rules. After every rule application that transforms the program, InvTS must reanalyze the transformed program. Thus, many applications of small rules are slower than a single application of one big rule, even when both produce the same result.

Correctness. We experimentally checked that the incrementalization preserved program semantics, using the same approach as in [71]. Our testing suite randomly generates a sequence of 50 million RBAC operations. It then verifies that the straightforward and incremental implementations produce the same output for these operations.

Performance testing of Constrained RBAC. For performance testing, we used a tool that generates random instructions based on potentially multiple dependent and independent variables. We then measured the time it takes to process these instructions. We performed two sets of experiments: One to show and contrast the effects of fully and partially incrementalizing often-called functions such as `CreateSession` and `AddActiveRole`; the other to verify that we do not degrade the performance of Core RBAC without constraints, and to show that the most-often called function, `CheckAccess`, is `O(1)` when incrementalized. Partially incrementalized refers to Constrained RBAC incrementalized by only the rules that we used in the past to incrementalize Core RBAC. Fully incrementalized refers to Constrained RBAC incrementalized by the entire library of 21 rules, including the 7 rules developed for incrementalizing Constrained RBAC.



(a) Original vs Complete



(b) Partial vs Complete

Figure 5.8: For Constrained RBAC, running times of `CreateSession` and `AddActiveRole`, 100000 repeats, 1 user/role.

When talking about the time and space complexity of Constrained RBAC, we use the following for sizes of the respective sets:

```
set:  OBJS OPS OPS*OBJS USERS ROLES SESSIONS SsdNAME C
size: O    A   P        U     R     S        N       C
```

where `A` for operation is adopted from the initial letter of "access" or "action", and `P` equals `A*O` and is the initial letter of "permission"; set `C` is the set of different cardinalities that appear in `SsdNC`. We use `x/y` to denote the number of `x`'s per `y`, where `x` and `y` are the letters for the above sets but in lower case. For example, `r/u` denotes the number of roles per user. For Constrained RBAC, `c/n` is 1 because, as per ANSI specification each name has one and only one constraint associated with it.

Table 5.3 shows the time complexities of `CreateSession`, `AddActiveRole`, and `CheckAccess`, computed using the method from [71]. To verify that our analyses of running time for `CreateSession` and `AddActiveRole` are correct, we performed two experiments. We vary the number of sessions that have a given role. We chose this variable as completely incrementalizing `CheckDSD` affects both `CreateSession` and `AddActiveRole`. Figure 5.8(a) shows that is indeed the case. The experiment varies the number of sessions from 10 to 100, with a step size of 10. The number of roles is fixed at 100. The experimental results confirm that the running time of the incrementalized version of `AddActiveRole` is constant in the number of sessions, and the running time of `CreateSession` decreases as the number of sessions increases. This contrasts with the straightforward versions, which are proportional to the number of sessions.

We then show the difference between partially and completely incrementalizing Constrained RBAC. Figure 5.8(b) shows that, as expected, the partially incrementalized versions of `AddActiveRole` and `CreateSession` are almost linear in the number of sessions. This is in stark contrast to the fully incrementalized versions, which are constant or better. This corresponds to the results obtained by complexity analysis.

We also verify that the invariant rules introduced to handle `CheckSSD`, `CheckDSD`, and other Constrained RBAC methods do not degrade the performance of the incrementalized program more than they have to. For this, we performed a subset of the same experiments performed in [71]. Specifically, we vary the number of Roles between 10 and 100, inclusive. We compare the performance of the most common function `CheckAccess`. The results confirm that the straightforward version does behave linearly in the number of roles, while the incrementalized version takes a constant amount of time.

5.6 Related work

We discuss existing work related to composition of invariant rules, BitTorrent, NetFlow, and Constrained RBAC.

Aspect-oriented programming. Aspect-oriented programming (AOP) [57, 55] also allows code for cross-cutting concerns, such as debugging, to be specified separately and inserted automatically at a set of matched program points. Connections between AOP and invariants are studied specially [95, 94]. Compared with existing AOP languages, InvTL has an explicit definition of invariant-preserving rules, to facilitate formal verification; it provides powerful static analysis, especially for automatically detecting updates, to coordinate transformations at queries and updates and minimize runtime overhead; finally, AOP does not help the programmer write code to efficiently maintain the `match` and `total` queries — he has to figure that out on his own.

Composition of program transformations. There are two approaches to sequential composition of program transformation specifications. In the extensional approach, the specifications are simply concatenated, separated by a sequential composition operator; applying the resulting specification to a program involves applying the original transformations, one at a time, in the specified order. In the intensional approach, the specifications are composed (merged) into a single transformation specification that can be applied in one shot. The extensional approach is used in StrategoXL and TXL. The intensional approach is used in J& [79], but is limited to specifications that do not depend on static analysis results. Our previous work [71, 42] also uses an extensional approach, but, unlike StrategoXL and TXL, it automatically determines the order for applying transformation rules. This chapter presents a method for intensional composition without the limitations of J&: we allow the invariant rules to depend on static analysis results; we also perform optimizations of the composed rule.

BitTorrent. While there are many sophisticated approaches that make BitTorrent more resilient to both network errors and outright malicious behavior, e.g. [19, 110, 59], these require either extensive manual modifications to a BitTorrent client, thus necessitating in-depth knowledge of the particular client, or significant expertise in network engineering. Our approach based on instrumentation rules and invariant rules allows us to modify a BitTorrent client to detect error-prone peers with only a passing familiarity with both the BitTorrent protocol and the implementation of the BitTorrent client. It is not difficult to extend our instrumentation rule for BitTorrent to modify the client to make it more resilient.

NetFlow. Design and implementation of NetFlow probes and collectors / analyzers that operate at line speeds on Gigabit links (100,000+ packets/sec) is challenging [28, 90]. On the collector / analyzer side, the challenge comes from the classic tension between clarity and efficiency, i.e., the desire to let the network administrator write analysis scripts in a declarative manner vs. the desire to have these scripts process hundreds of thousands of packets per second. Some systems allow a degree of customization of the queries they efficiently execute [27, 96]. Our technique potentially allows such systems to execute even more general queries efficiently by applying invariant rules to such queries, and performing automated composition and optimization of complex invariant rules.

Constrained RBAC. Various implementations of Constrained RBAC exist, such as [103, 100, 34]. We are aware of only one incrementalized implementation — Strembeck's, and it was incrementalized manually, with the associated difficulties of verifying its correctness.

BitTorrent

	# LOC before	# LOC after	#rules	composition time (s)	optimization time (s)	**application time (s)**	**notifications stored (mil)**	**extra set ops. (mil)**	CPU usage	total network
Original	41,162	41,162	-	-	-	-	-	-	48.6%	32.1GB
Separate appl.	41,162	41,374	6	-	-	2998	96.3	193.3	56.1%	33.1GB
Composed appl.	41,162	41,374	6	2.9	-	2320	93.1	189.6	56.9%	32.7GB
Optimized comp.	41,162	41,331	6	2.8	3.5	2261	25.0	58.8	49.1%	33.3GB

NetFlow query tool

	# LOC before	# LOC after	#rules	composition time (s)	optimization time (s)	**application time (s)**	**total proc. time (s)**	**throughput (packets/s)**
Original query	64	64	-	-	-	-	>600	81
Separate appl.	64	105	5	-	-	21.1	33.1	302,114
Composed appl.	64	105	5	1.0	-	15.3	32.8	304,878
Optimized comp.	64	75	5	1.0	0.4	15.4	19.9	502,512

Constrained RBAC

	# LOC before	# LOC after	#rules	composition time (s)	optimization time (s)	**application time (s)**	# `inv` clauses applied
Separate appl.	381	2,183	21	-	-	257.4	38
Composed appl.	381	2,183	21	1.1	-	44.2	27
Optimized comp.	381	2,183	21	1.1	0.5	44.8	27

Table 5.2: Summary of rule composition experiments. For each experiment, we give the size before and after rule application, the number of rules, and the time InvTS took to compose, optimize, and apply the rules. For BitTorrent, we give the number of notifications stored, the number of additional set operations performed, the CPU usage, and the total network usage. For NetFlow, we give the time to process 10 million packets, and the number of packets processed per second. For Constrained RBAC, we give the number of InvTS `inv` clauses applied.

functionalties	straightforward	inc Core RBAC queries	inc all queries
`CreateSession`(u,s,ars)	ars+R+S*N*R	p/r*(ars+r/s)+S*N*s/r	p/r*(ars*n/r+r/s)+N
`AddActiveRole`(u,s,r)	R*S*N*R	p/r*S*N*r/s	p/r+n/r
`CheckAccess`(s,op,o)	R	1	1

Table 5.3: Time complexities of selected Constrained RBAC functionalities.

Chapter 6

Conclusion

This dissertation presents a system that allows coordinated transformations driven by invariants to be specified declaratively, as invariant rules, and applied automatically. It describes the system's implementation for efficiently applying invariant rules to Python and C programs; it also describes the type, control flow, and alias analyses developed for applying the invariant rules in a coordinated manner, and in a way that does not introduce excessive overhead to the transformed program. To illustrate the use of the transformation system, we describe successful applications of this system in generating efficient implementations from clear and modular specifications, in instrumenting programs for runtime invariant checking, query-based debugging, profiling, and refactoring. We also describe a method for composing and optimizing invariant and instrumentation rules that enables a large class of rules to both be more efficiently applied, and result in faster transformed programs; applying this method to transformations for instrumentation of BitTorrent and a NetFlow query tool, and to incrementalization of Constrained RBAC shows reduced transformation time and significantly decreased running time and memory overhead of the instrumentation.

Future work. Manually creating correct incrementalization rules is challenging. To combat this, future work in the direction of automated incrementalization could be the integration of InvTS and a method developed by Rothamel [87] that can, for a large class of queries, automatically generate invariant rules; another approach would be the development of methods and tools for verification of invariant rules.

On the implementation side, there are two clear directions for future work: (1) the static analyses required by InvTS are expensive, and (2) the `cost` clause is not implemented in the Python LM. Future work would include further refinement and optimization of static analyses, and the implementation of the `cost` clause.

Another promising direction is to extend the runtime invariant checking framework that we presented in this dissertation to not just check a given invariant at runtime, but to detect possible invariants by instantiating all possible invariants that match a given pattern and detecting which ones hold at runtime.

Bibliography

[1] U.A. Acar. Self-adjusting computation:(an overview). In *Proc. of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pages 1–6. ACM, 2009.

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

[3] American National Standards Institute, Inc. Role-Based Access Control. ANSI INCITS 359-2004, 2004. Approved Feb. 3, 2004.

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

[5] 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.

[6] C. Artho, V. Schuppan, A. Biere, P. Eugster, M. Baur, and B. Zweimüller. JNuke: efficient dynamic analysis for Java. *Lecture Notes in Computer Science*, 3114:462–465, 2004.

[7] 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.

[8] 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.

[9] M. Barnett, B.Y.E. Chang, R. DeLine, B. Jacobs, and K.R.M. Leino. Boogie: a modular reusable verifier for object-oriented programs. In *Proc. of the 4th Intl. Symp. on Formal Methods for Components and Objects*, pages 364–387, 2006.

[10] M. Barnett, R. DeLine, M. Fahndrich, K.R.M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.

[11] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *Proc. of the 5th Intl. Conf. on Verification, Model Checking and Abstract Interpretation*, pages 44–57, 2004.

[12] D. Bartetzko, C. Fischer, M. Möller, and H. Wehrheim. Jass Java with assertions. *Electronic Notes in Theoretical Computer Science*, 55(2):103–117, 2001.

[13] F.L. Bauer, B. Möller, H. Partsch, and P. Pepper. Formal program construction by transformations — computer-aided, intuition-guided programming. *IEEE Transactions on Software Engineering*, 15(2):165–180, 1989.

[14] M. Bravenboer, K.T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Science of Computer Programming*, 72(1-2):52–70, 2008.

[15] S. Callanan, D. J. Dean, and E. Zadok. Extending GCC with modular GIMPLE optimizations. In *Proc. of the 2007 GCC Developers' Summit*, pages 31–38, 2007.

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

[17] Y. Cheon. *A Runtime Assertion Checker for the Java Modeling Language.* PhD thesis, Iowa State University, 2003.

[18] 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.

[19] A.L. Chow, L. Golubchik, and V. Misra. Improving BitTorrent: a simple approach. In *Proc. of the 7th Intl. Workshop on Peer-to-Peer Systems*, 2008.

[20] B. Claise. Cisco Systems NetFlow services export version 9. RFC 3954, Internet Engineering Task Force, Oct. 2004.

[21] L.A. Clarke and D.S. Rosenblum. A historical perspective on runtime assertion checking in software development. *ACM SIGSOFT Software Engineering Notes*, 31(3):25–37, 2006.

[22] R. Cole, R. Hariharan, and P. Indyk. Tree pattern matching and subset matching in deterministic O($n\ log^3 n$)-time. In *Proc. of the 10th ACM-SIAM Symp. on Discrete Algorithms*, pages 245–254, 1999.

[23] J. R. Cordy, T. R. Dean, A. J. Malton, and K. A. Schneider. Software engineering by source transformation-experience with TXL. In *Proc. of the 1st IEEE Intl. Working Conf. on Source Code Analysis and Manipulation*, pages 170–180, 2001.

[24] J.R. Cordy. TXL-a language for programming language tools and applications. *Electronic Notes in Theoretical Computer Science*, 110:3–31, 2004.

[25] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of the 4th ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages*, pages 238–252, 1977.

[26] C. Delord. TPG: a Python Toy Parser Generator.

[27] L. Deri. Passively monitoring networks at Gigabit speeds using commodity hardware and open source software. In *Proc. of the Passive and Active Measurement Conf.*, pages 13–21, 2003.

[28] L. Deri and V. Matteucci. nProbe: an open source netflow probe for gigabit networks. In *Proc. of the 2003 TERENA Networking Conf.*, pages 1–4, 2003.

[29] Arie Van Deursen, Jan Heering, and Paul Klint, editors. *Language Prototyping: An Algebraic Specification Approach: Vol. V.* World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1996.

[30] 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.

[31] 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.

[32] 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.

[33] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed NIST standard for role-based access control. *ACM Transactions on Information and System Security*, 4(3):224–274, 2001.

[34] T. Finin, A. Joshi, L. Kagal, J. Niu, R. Sandhu, W. Winsborough, and B. Thuraisingham. R OWL BAC: representing role-based access control in OWL. In *Proc. of the 13th ACM Symp. on Access Control Models and Technologies*, pages 73–82, 2008.

[35] 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.

[36] 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.

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

[38] D. Gopan and T. Reps. Lookahead Widening. In *Proc. of 18th Conf. on Computer-Aided Verification*, pages 452–466, 2006.

[39] M. Gorbovitski. A survey of program transformation languages and systems. `http://public.zavulon.com/publications/RPE.pdf`.

[40] M. Gorbovitski, Y. A. Liu, S. D. Stoller, and T. Rothamel. Composing transformations for instrumentation and incrementalization of real applications. Preliminary technical report, Computer Science Department, SUNY Stony Brook, 2010.

[41] M. Gorbovitski, Y. A. Liu, S. D. Stoller, K. T. Tekle, and T. Rothamel. Alias analysis for optimization of dynamic languages. In *Proc. of the 2010 Dynamic Languages Symp.*, pages 12–20, 2010.

[42] 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 Intl. Workshop on Dynamic Analysis*, pages 43–49, 2008.

[43] 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.

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

[45] D. Goyal and R. Paige. The formal reconstruction and improvement of the linear time fragment of willard's relational calculus subset. In *Proc. of the 1997 IFIP TC2 Working Conference on Algorithmic Languages and Calculi*, pages 382–414, 1997.

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

[47] Matthew A. Hammer, Umut A. Acar, and Yan Chen. Ceal: a c-based language for self-adjusting computation. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 25–37, New York, NY, USA, 2009. ACM.

[48] 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.

[49] K. Havelund and G. Rosu. An Overview of the runtime verification tool Java PathExplorer. *Formal Methods in System Design*, 24(2):189–215, 2004.

[50] 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.

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

[52] International Organization for Standardization. Z formal specification

notation — Syntax, type system and semantics. ISO/IEC 13568:2002.

[53] 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.

[54] Rajesh Kazhankodathed. http://tinyurl.com/5b9qfe.

[55] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *Proc. ECOOP 2001, LNCS 2072*, pages 327–353, Berlin, June 2001. Springer-Verlag.

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

[57] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proceedings of the 11th Europeen Conference on Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, 1997.

[58] M.Z. Kim, M. Viswanathan, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: a run-time assurance approach for Java programs. *Formal Methods in System Design*, 24(2):129–155, 2004.

[59] M. A. Konrath, M. P. Barcellos, and R. B. Mansilha. Attacking a swarm with a band of liars: evaluating the impact of attacks on BitTorrent. In *Proc. of the 7th IEEE Intl. Conf. on Peer-to-Peer Computing*, pages 37–44, 2007.

[60] B. Krause and T. Wahls. jmle: a tool for executing JML specifications via constraint programming. *Lecture Notes in Computer Science*, 4346:293–296, 2007.

[61] V. Kuncak and M. Rinard. An overview of the Jahob analysis system: project goals and current status. In *Proc. of the 20th Intl. Parallel and Distributed Processing Symp.*, pages 8–16, 2006.

[62] 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.

[63] 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.

[64] G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. *Science of Computer Programming*, 55(1-3):185–208,

2005.

[65] G.T. Leavens, A.L. Baker, and C. Ruby. Preliminary design of JML: a behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, 2006.

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

[67] R. Lencevicius, U. Hölzle, and A.K. Singh. Dynamic query-based debugging of OO programs. *Automated Software Engineering*, 10(1):39–74, 2003.

[68] S. Lerner, T. Millstein, E. Rice, and C. Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. In *Proc. of the 32nd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 364–377, 2005.

[69] 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.

[70] Y. A. Liu and S. D. Stoller. Role-based access control: A simplified specification. Technical Report DAR 05-24, Computer Science Department, SUNY Stony Brook, Aug. 2005 (Revised Jan. 2006).

[71] Y. A. Liu, C. Wang, M. Gorbovitski, T. Rothamel, Y. Cheng, Y. Zhao, and J. Zhang. Core role-based access control: Efficient implementations by transformations. In *Proc. of the 2006 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pages 112–120, 2006.

[72] 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.

[73] 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.

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

[75] S. McPeak. Elsa C++ Frontend, 2008.

[76] 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.

[77] 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.

[78] P. Moor. Free Riding in BitTorrent and Countermeasures. *Master's The-*

sis, Distributed Computing Group, Computer Engineering and Networks Laboratory, Swiss Federal Institute of Technology Zurich, Summer, 2006.

[79] N. Nystrom, X. Qi, and A.C. Myers. J&: Nested intersection for scalable software composition. In *Proc. of the 21st Annual ACM SIGPLAN Conf. on Object Oriented Programming Systems, Languages, and Applications*, pages 21–36, 2006.

[80] R. Paige. Viewing a program transformation system at work. In *Proc. of the 6th Intl. Symp. on Programming Language Implementation and Logic Programming*, pages 5–24, 1994.

[81] H. Partsch and R. Steinbrüggen. Program transformation systems. *ACM Comput. Surv.*, 15(3):199–236, 1983.

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

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

[84] 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.

[85] J.W. Robert and K. Viggers. Implementing protocols via declarative event patterns. *ACM SIGSOFT Software Engineering Notes*, 29(6):1–21, 2004.

[86] S. Romig. The OSU flow-tools package and Cisco NetFlow logs. In *Proc. of the 14th USENIX Conf. on System Administration*, page 304, 2000.

[87] T. Rothamel. *Automatic Incrementalization of Quries in Object-Oriented Programs*. PhD thesis, Computer Science Department, SUNY Stony Brook, 2008.

[88] 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.

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

[90] F Schneider. *Performance evaluation of packet capturing systems for high-speed networks*. PhD thesis, Munich Technical University, 2005.

[91] A. Shankar and R. Bodík. DITTO: automatic incrementalization of data structure invariant checks (in Java). In *Proc. of the 2007 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 310–319, 2007.

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

[93] D. R. Smith. KIDS: A semiautomatic program development system.

IEEE Transactions on Software Engineering, 16(9):1024–1043, Sept. 1990.

[94] D. R. Smith. Requirement enforcement by transformation automata. In *Proc. of the 6th Workshop on Foundations of Aspect-Oriented Languages*, pages 5–14, 2007.

[95] D. R. Smith. Aspects as invariants. *Automatic Program Development: A Tribute to Robert Paige*, pages 270–286, 2008.

[96] SolarWinds. Orion NetFlow Traffic Analyzer, June 2009.

[97] J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice-Hall, 2nd edition, 1992.

[98] 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.

[99] 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.

[100] M. Strembeck. Conflict checking of separation of duty constraints in RBAC — implementation experiences. In *Proc. of the 2004 Intl. Conf. on Software Engineering*, pages 224–229, 2004.

[101] Program-Transformation.org Team. Program-transformation.org - the program transformation wiki. `http://www.program-transformation.org/`.

[102] M.G.J. van den Brand, H.A. de Jong, P. Klint, and P.A. Olivier. Efficient annotated terms. *Software, Practice and Experience*, 30(3):259–291, 2000. `citeseer.ist.psu.edu/vandenbrand00efficient.html`.

[103] M. Ventuneac, T. Coffey, and I. Salomie. A policy-based security framework for web-enabled applications. In *Proc. of the 1st Intl. Symp. on Information and Communication Technologies*, pages 487–492, 2003.

[104] E. Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. *Lecture Notes in Computer Science*, 3016:216–238, 2004.

[105] E. Visser. A survey of strategies in rule-based program transformation systems. *Journal of Symbolic Computation*, 40(1):831–873, 2005.

[106] 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.

[107] D. E. Willard. Quasilinear algorithms for processing relational calculus expressions (preliminary report). In *Proc. of the 9th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pages 243–257, 1990.

[108] D. Willis, D. J. Pearce, and J. Noble. Caching and incrementalisation in

the java query language. In *Proc. of the 23rd Annual ACM SIGPLAN Conf. on Object Oriented Programming, Systems, Languages, and Applications*, pages 1–18, 2008.

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

[110] K. Y. Wong, K. H. Yeung, and Y. M. Choi. Solutions to swamp poisoning attacks in BitTorrent networks. In *Proc. of the 2009 Intl. MultiConf. of Engineers and Computer Scientists*, pages 360–363, 2009.

[111] Wuu Yang. Identifying syntactic differences between two programs. *Software - Practice and Experience*, 21:739–755, 1991.

[112] K. Zee, V. Kuncak, M. Taylor, and M. Rinard. Runtime checking for program verification. In *Proc. of the 7th Intl. Workshop on Runtime Verification*, pages 202–213, 2007.

[113] C. Zhao, Y. Chen, D. Xu, N. Heilili, and Z. Lin. Integrative security management for web-based enterprise applications. In *Proc. of the 6th Intl. Conf. on Web-Age Information Management*, pages 618–625, 2005.