Improving the Precision of Equality-Based Dataflow Analyses[*]

Erik Ruf

Microsoft Research
One Microsoft Way
Redmond, WA 90852-6399 USA
`erikruf@microsoft.com`

Abstract. We present two new, orthogonal techniques for improving the precision of equality-based dataflow analyses. *Subtype expansion* models objects at a per-type granularity, enabling a form of subtype-restricted equality constraint, while *mutation tracking* uses a simple effect analysis to avoid a class of false aliases induced by the bidirectional nature of equality constraints. The utility and costs of these techniques are demonstrated in a context-sensitive interprocedural optimization whose static precision improves by 6-600% when our techniques are applied.

1 Introduction

Equality-based dataflow analyses have become increasingly popular due to their simplicity and efficiency. Such techniques provide flow-insensitive information at a low cost in implementation effort and analysis execution time. Applications include binding time analysis [Hen91], pointer analysis [Ste96b,Ste96a,LH99], escape analysis [Ruf00], call graph construction [GDDC97], synchronization elimination [Ruf00], and thread-specific storage allocation [Ste00].

The benefits of equality-based techniques are offset by a loss of precision with respect to other flow-insensitive techniques such as set-based analysis. A variety of research has addressed this issue. [Ste96a] explicitly models field values within record data structures. Interprocedural context sensitivity has been added both by instantiation of procedure summaries at call sites [LH99,Ruf00] and by deferred traversal of explicit instantiation constraints at query time [FRD00]. [Das00] models assignment with a combination of equality and unidirectional flow constraints, while [DGC98] limits the indegree and outdegree of nodes involved in unidirectional constraints by dynamically switching to equality constraints.

This paper describes two new techniques for achieving additional precision in equality-based systems:

[*] This document is a preprint of an article to appear in Static Analysis Symposium (SAS) 2002, and is copyright © Springer-Verlag.

– **Subtype expansion** models individual objects at a per-runtime-type granularity, avoiding overly conservative equality constraints in the face of program type restrictions (static typing, dynamic up- and downcasting, virtual function binding).
– **Mutation tracking** distinguishes between reads and writes of objects reachable from formal parameters, avoiding the propagation of a class of false (infeasible) aliases from procedure summaries to call sites.

These techniques are largely orthogonal to one another and to the prior work referenced above. We demonstrate this by describing and assessing our techniques in the context of a system whose base interprocedural framework configuration performs field modeling and procedure summary instantiation. For a sample optimization that removes unnecessary downcast checks, the static and dynamic fractions of total checks removed are improved by 6-600% and 0-4400%, respectively.

2 Background

The techniques described in this paper are implemented in the Marmot native compilation system for Java [FKR^{+}00]. Marmot implements most Java 1.1 semantics and libraries [GJS96], but does not support dynamic loading and limits the use of reflection. The resulting "closed-world" assumption enables a number of whole-program analyses, including an equality-based, flow-insensitive, context-sensitive analysis with method specialization [Ruf00]. This section briefly summarizes relevant aspects of this platform, which we will call the *base analysis.*

Runtime values of reference type (objects) are modeled by a union-find data structure called an *object representative*[1] that contains a list of attributes (arbitrary elements of dataflow lattices) and a mapping from field names to object representatives that represent field and array element values. Some attributes, such as *static* (indicating that the modeled runtime values may be transitively reachable from a static field), are recursively applied to an object reference's fields, while others, such as *synchronized* (indicating that an object may be the target of monitor lock/unlock operations) are not.

Intraprocedural analysis consists of traversing a method body in the context of a local symbol table that maps formal and local variables to new object representatives. The traversal evaluates expressions (which may create, dereference, or add attributes to object representatives), assignments (unifying [ASU86] the object representatives for the left and right hand sides), and method invocations. Nonrecursive invocations are modeled by a *mapping* algorithm similar to type instantiation (see Figure 1) that propagates aliases and attributes from the target method summary to the call site. Recursive invocations simply unify formal/actual, return/result, and thrown/caught representatives.[2]

[1] Object representatives were called *alias sets* in [Ruf00].
[2] Optionally, we could obtain additional precision by also using the mapping algorithm at recursive call sites. Doing so would require iteration over such sites in each

```
mapValues(summary, site) {
  memoTable = new MemoTable
  for (i=0; i<|summary|; i++)
    map(summary[i], site[i], memoTable)
}
map(repSum, repSite, memoTable) {
  if (repSum.isStatic)
    unify(repSum, repSite)
  else if (memoTable.containsKey(repSum))
    existing = memoTable[repSum]
    unify(existing, repSite)
  else
    memoTable[repSum] = repSite
    repSite.addAttribs(repSum.attribs)
    foreach (fieldname fn in repSum.fieldMap.keys)
      if (repSite.fieldMap[fn]==null)
        repSite.fieldMap[fn] = new ObjRep
      map(repSum.fieldMap[fn], repSite.fieldMap[fn], memoTable)
}
```

Fig. 1. Pseudocode for mapping procedure

Interprocedural analysis performs two walks of a topologically sorted strongly connected component (SCC) decomposition of a conservative static call graph. In the upward (callee→caller) propagation phase, the intraprocedural analysis is applied to each method in the current SCC, after which a signature, or *method summary* is extracted from each method's symbol table. An optional *pruning* operation compresses each summary by removing object representatives that cannot induce aliasing or attribute propagation at call sites. No fixed point iteration is required because recursively dependent summaries share object representatives.

The downward (caller→callee) interprocedural analysis is similar, except that the direction of propagation is reversed, and only attributes, not aliases, are propagated. After all of a method's call sites are processed, its dataflow information is complete, enabling transformation. Some versions of our analysis also perform *method specialization*, in which method signatures and bodies are copied to enable call-site-specific transformations.

Because the mapping procedure does not employ k-limiting, the analysis could consume exponential space and time with respect to program size. However, given that few large recursive data structures are constructed without it-

call-graph SCC under an extended-occurs-check predicate [Hen93]. Our preliminary experiments with such an approach have experienced scalability issues in the analysis of large SCCs.

eration or recursion, neither of which generate additional object representatives, this is unlikely in practice.

3 Subtype Expansion

3.1 Motivation

Our first technique focuses on the granularity at which the analysis models objects. In the base system, runtime objects are represented by static object representatives consisting of list of attributes describing the object and a mapping from fieldnames to object representatives. All forms of merging, including local assignment, exception dispatch, and the interprocedural value mapping that occurs at call sites, are implemented via unification[3] of these object representatives. Similarly, all forms of attribute assignment, such as marking an object as escaping or as having a particular runtime type, affect the attribute list(s) of their argument object representative(s).

This monolithic representation becomes overly conservative when subtyping is introduced. In an object-oriented language, both explicit (e.g., downcasting, `instanceof`) and implicit (e.g., virtual function and exception dispatch) operations can restrict the runtime types of objects. This restriction can be used to limit the scope of merging or attribute assignment involving such objects.

```
static Object myToString(Object o) {
  S1: return o.toString();
}
```

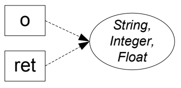


Fig. 2. Virtual call example. The base analysis infers an alias between `o` and the value returned by `S1`.

Figure 2 shows a simplified version of a case that arose in [Ruf00]. During the process of constructing a summary for method `myToString`, a virtual call with selector `Object.toString` and a receiver argument of static type `Object` is detected. Given that the argument's dynamic type is bounded only by the type `Object`, all known summaries of `toString` are applied. Most such applications are benign; e.g., `Integer.toString` and `Float.toString` merely return new string objects that point to new character arrays. However, `String.toString` is specified [GJS96] to return its receiver argument (no copying allowed), so applying it causes the object representatives for argument `o` and return value `r` to be unified, thus aliasing the receiver and result values, even when the receiver

[3] Our implementation performs unification and/or mapping as the merging operations are processed. However, our exposition applies equally well to systems that construct explicit constraints to be solved later.

```
Object G = ... //escapes
Object x = ...
if (...)
  S2: G.f = (T1)x;
else
  Object y = (T2)x;
```

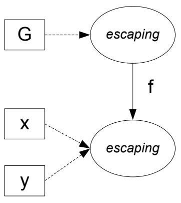


Fig. 3. Escape example. The base analysis marks both `x` and `y` as escaping.

is not a `String`. This would make it impossible, for example, to remove the dynamic type check in the expression `(String)myToString(o)`, even though `myToString` can only return strings.

Similar issues can arise in purely intraprocedural scenarios, and in analyses whose purpose is other than type propagation. For example, in Figure 3, if types `T1` and `T2` are unrelated in the class hierarchy, the fact that `(T1)x` escapes should not cause `y` to escape, but (under the base analysis) it does.

3.2 Solution

The common theme in the above examples is that multiple runtime objects described by a single analysis-time object representative may reach different operations depending on their [the objects'] runtime types. For example, only string objects `o` will cause an invocation of `String.toString` at (Fig. 2, `S1`), and only objects of type T1 will escape at (Fig. 3, `S2`).

Subtype expansion models this behavior by changing the granularity at which runtime objects are statically modeled, allowing an object representative to expose multiple union-find data structures.

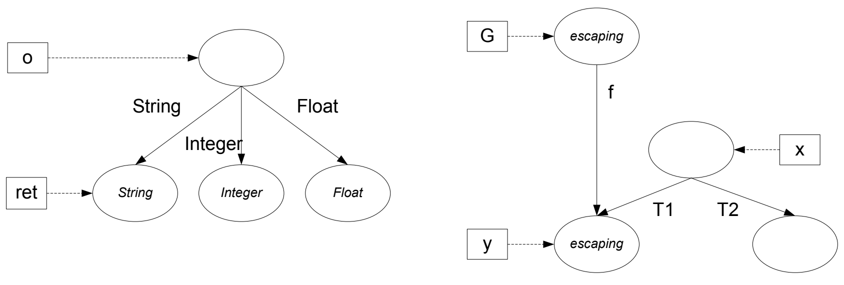
In one maximally-general implementation, each top-level object representative carries a vector of at most $|\mathcal{T}|$ sub-representatives, where $\mathcal{T}$ is the set of all reference types in the program. Each sub-representative models the object's behavior (i.e., carries attributes and a field mapping) at a set of types $\mathcal{T}_i \in \mathcal{T}$ where all $\mathcal{T}_i$ are disjoint and $\bigcup_i \mathcal{T}_i = \mathcal{T}$.[4] Unification of object representatives o_1 and o_2 "at" a type $T \in \mathcal{T}_i$ consists of unifying the sub-representatives $o_1.[\mathcal{T}_i]$ and $o_2.[\mathcal{T}_i]$as well as all pairs of object references $(r_1.\mathit{fields}[f], r_2.\mathit{fields}[f])$ where T declares or inherits a field f. Such a representation allows for arbitrary type-based restriction of value flow. For example, it can precisely represent $\{t \mid t \not\leq T\}$, which could be imposed on the type of `y` in a `catch` clause guarding `x = (T)y`.

Our prototype implementation achieves a lower space cost by precisely modeling only restrictions to subtype hierarchies (type ideals) rather than arbitrary sets of types. Our representation mirrors the the underlying type hierarchy. Each

[4] The partitioning of $\mathcal{T}$ into subsets $\mathcal{T}_i$ can take place statically or dynamically. In the static case, maximal precision is obtained by choosing $\mathcal{T}_i$ of size 1. In the dynamic case, precision is maintained by splitting existing subrepresentatives when a finer-grained restriction is required.

object representative is associated with a single reference type R, and carries an attribute list, a field mapping, and a "subtype" mapping (from immediate subtypes of R to object representatives). Field mappings in "subtype" representatives share state (mappings for inherited fields) with those of their supertypes.[5] Each runtime value is modeled by an object representative whose associated type is `Object`. Unifying two such representatives at a type T consists of unifying the subtype representatives associated with T, which will induce pairwise unification of representatives of fields directly implemented by types $T' \geq T$.

The implementation of attributes under subtype expansion differs slightly from that under the base analysis (as described in Section 2). Transitive reachability attributes such as *static* now recurse over both field and subtype representatives, while object-level attributes such as *synchronized* recurse over subtype representatives. A third variety of attribute that applies only to a single object representative can be used to represent information that is relevant only at a particular type such as a boolean flag that models whether the represented object is allocated at a particular type.



(virtual call example)

(escape example)

Fig. 4. Example representations with subtype expansion

Figure 4 shows the representation of the example programs in Figures 2 and 3 when subtype expansion is used. In the virtual call example, only string-valued instances of `o` can be returned, even though `o`'s static type is `Object`. In the escape analysis example, only subtypes of `T1` escape, despite `x`'s declaration at type `Object`.

The primary costs of subtype expansion arise from the increased size of object representatives and the increased number of unification operations required. In the asymptotic worst case, these may be multiplied by the size of the class

[5] This implementation can be viewed as being "factored" with respect to subtypes, but not with respect to fields. A more efficient representation might avoid this duplication; e.g., by making field maps first-class, unifiable datatypes and allowing a subtype representative to reference its parent's field map.

hierarchy and (in our implementation) the maximum number of fields visible to a class.

4 Mutation Tracking

4.1 Motivation

```
p (f1, f2, f3, f4, f5) {
  S1: f1.f = f2;
  S2: x = (...) ? f3.g : f4.h;
  S3: synchronized(x) { ... }
  S4: return f5
}
r = p(a1, a2, a3, a4, a5);
```

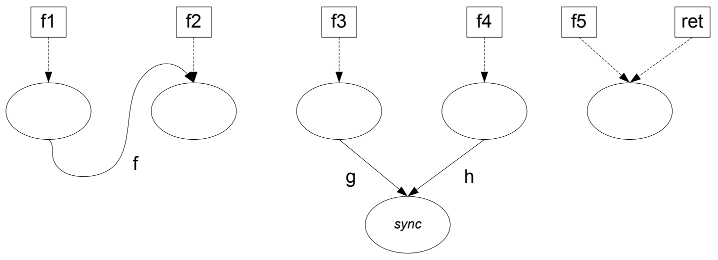


Fig. 5. False alias example

Our second technique addresses the bidirectional nature of equality constraints. Within procedures, a typical equality-based analysis ensures that any two object references that could represent the same runtime value constrained to be equal. Interprocedurally, it ensures that call-site object references (and their attributes) are instances of the corresponding references in the callee. This can result in unwanted aliasing.

Consider a call to procedure `p`, whose code and method summary are shown in Figure 5. Under our base analysis, the alias created in statement `S1` is reflected at the call site by making `a2`'s representative reachable from that of `a1` (via field f). Similarly, `S3` causes the object reference associated with `a5` to be unified with that of `r`. These operations are sufficient to represent (at the call site) the aliasing effects of calling `p`. However, the analysis also unifies the representatives of `f3.g` and `f4.h` because they appear in a common context, namely `S2:x`, thus ensuring the existence of a common representative for `a3.g` and `a4.g`, marked with the "synchronized" attribute. In fact, only the attribute need be propagated—the call does not induce an alias between `a3.g` and `a4.h`.

A more complex situation arises with container objects such as `Vector` and `Hashtable`, instances of which could contain themselves, thus causing the representative for `this` to be unified with those of the elements of the array(s) used to implement the container. For example, invocations of `Hashtable.get` and `Hashtable.equals` cause the analysis to infer caller-side aliases between the table, its keys, and its values, even though neither procedure mutates the table.

4.2 Solution

Mutation tracking avoids false actual/actual aliases by distinguishing between two kinds of equality constraints. Some constraints, such as that inferred for statement `S1` in Figure 5, add a potentially new path (alias) from one caller-side object to another. For correctness, such constraints must be reflected to the call site when the method summary is applied. Other equality constraints arise merely because multiple formal-reachable representatives reach a common context (e.g., Figure 5, statement `S2`). While such constraints are necessary for the propagation of attributes, they do not induce any new paths between caller objects and thus should not induce caller-side constraints.

Our implementation of mutation tracking distinguishes these cases by adding an optional *mutation* attribute to the object representative data structure. Mutation attributes are recursively propagated across child fields and (if subtype expansion is used) child types. Whenever the intraprocedural analysis encounters a field or array element update, it sets the mutation attribute of the representative for the new value, representing the fact that the new path(s) represented by the resulting unification operations arose due to mutation rather than context sharing.

Mutation attributes are used to limit alias propagation at call sites. This requires changes to the processing of procedure invocations. At nonrecursive call sites, the mapping process of Figure 1 is changes as follows: a callee representative f is added to the memo table only when it

1. is reachable from the return or thrown representative, or
2. transitively reaches a representative having the mutation attribute

Note that even when a representative is not added to the memo table, its attributes are still propagated.

In the example of Figure 5, the representative shared by `f1.f` and `f2` will have the mutation bit set, forcing unification of `a1.f` and `a2`, with the result having the mutation attribute. The representatives for `r` and `a5` are unified because `f5`'s representative is returned. The representative for `x` is neither returned nor marked as a mutation, As shown in Figure 6, the representatives for `a3.g` and `a4.g` are both marked with the synchronized attribute, but are not unified.

The treatment of recursive calls need not be changed; unifying corresponding caller and callee object representatives as before is sufficient. Doing so preserves the property that no fixed point iteration is required, but may result in the propagation of some false aliases within strongly connected components of the call graph.

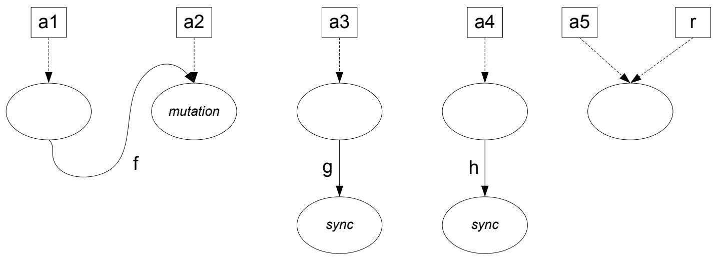


Fig. 6. False alias example with mutation tracking

Alternatively, the analysis can use the same criteria for justifying unification at recursive calls as is used for justifying memo table insertion at nonrecursive calls. However, unlike the nonrecursive case, here the algorithm must make its decision with incomplete knowledge, as the analysis of the callee procedure may be incomplete. This requires iteration over all recursive calls in the current SCC until no more call-site unifications can be performed. We have implemented and assessed both of the above techniques.

For nonrecursive calls, the additional tests (1) and (2) above can either be computed in constant time or are already performed as part of the pruning process. In the case of recursive calls, item (2) requires an additional search of the formal representatives. If required, iteration only reprocesses formal/actual pairs where unification has been deferred.

5 Assessment

5.1 Sample analysis

We tested our improvements on a simple prototype analysis that models runtime types of objects. Type propagation is an interesting application for equality-based techniques because (unlike the more common application, escape analysis) it is a unidirectional problem, in which the bidirectional nature of equality-based techniques leads to additional imprecision (often called "backflow").

For the sake of simplicity, we use the analysis to transform only explicit downcast operations, and do not perform specialization on the downward pass. All versions of the analysis make use of the intermediate representation described in Section 3.2.[6] On the upward pass, constants and objects returned by constructors carry their runtime type in the form of an *allocated* annotation on the object representative for the appropriate subtype of `Object`. On the downward pass, each explicit downcast test (on type T) is checked: if all allocated annotations on

[6] This choice improves the precision of our assessment by avoiding accidental differences introduced by representation changes, but does exact a price in analysis time for versions not making use of the subtype expansion capability.

the operand's (transitive) "subtype" children are on object references for types $T' \leq T$, the test is removed.

5.2 Benchmarks

Table 1. Benchmark programs

name	meths	stmts	checks	
			static	dynamic
`_213_javac`	2091	33319	321	6.41E+06
`_202_jess`	1541	20703	95	1.51E+07
`_209_db`	799	9268	42	5.32E+07
`_228_jack`	1043	17825	122	5.70E+06
`jlex100`	637	11413	67	1.14E+08
`javacup`	948	17786	434	1.49E+06

We tested our techniques on six programs (described in Figure 1) chosen because they either perform a large number of downcast check operations or make extensive use of object-polymorphic container types.

The analysis and optimization were implemented in the Marmot [FKR+00] optimization pipeline immediately following initial parsing and cleanup, but before inlining, local type propagation, etc. Marmot is a whole-program compiler; the optimization was applied to the user program and most non-native library methods (a few methods involving low-level synchronization, etc. were explicitly modeled and left untransformed).

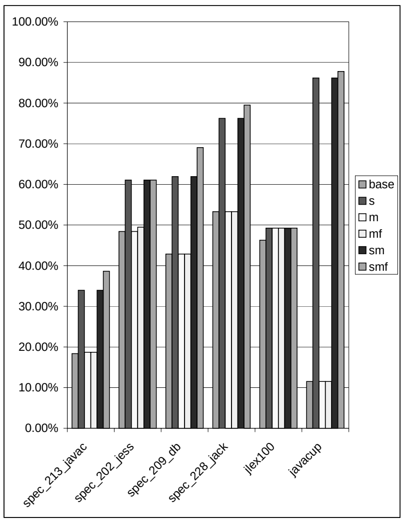
We tested six distinct configurations of the prototype. The "base" configuration enables the analysis without without extensions. The remaining configurations selectively enable combinations of three techniques: "s" denotes subtype expansion, "m" denotes mutation tracking, and "f" denotes fixed point iteration.

All testing was performed on a 1.7GHZ dual-processor Intel Xeon processor with 1GB of RAM under Windows XP Professional, using the Marmot generational copying collector with a 256MB heap. All of the benchmark programs executed deterministically, so all operation counts are precise; timings are the average of multiple executions.

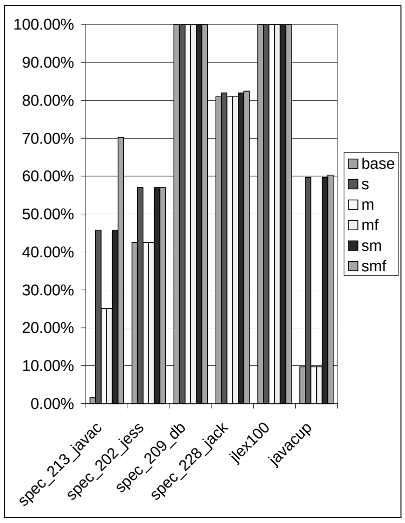
5.3 Precision

Figure 7 shows the fraction of static and dynamic downcast checks for each benchmark program and configuration.[7] With the exception of `jlex100`, our techniques removed at least 10% more static checks than the base analysis, with subtype expansion responsible for the majority of the gain. Mutation tracking

[7] Absolute data are available in the Appendix.



(static)



(dynamic)

Fig. 7. Fraction of cast check operations removed

improved upon the base analysis slightly on `javac` and `jlex100`; iterated mutation tracking improved `jess`. Adding mutation tracking to subtype expansion had little effect unless iteration was enabled, where it had significant effects on four of six programs.

The improved static precision had little to no dynamic effect in the programs `db`, `jack`, and `jlex`, where the base analysis already achieved excellent results. In the remaining programs, substantial improvements were obtained, with an additional 15-69% of dynamic checks removed.

Given that the sample analysis emphasized type precision rather than alias effects (which would be more important in, e.g., an escape analysis), it is not entirely surprising that most of the measured improvement came from subtype expansion. More interesting, perhaps, is that mutation tracking exposed additional opportunities only when iteration was used, suggesting that false aliases at potentially recursive call sites are more problematic than those at nonrecursive sites. This effect may be due to the use of dynamic object-based polymorphism to implement homogeneous containers (vectors and hash tables) where static parametric polymorphism would suffice.

5.4 Execution times

The primary purpose of this study was to evaluate the precision improvements available via our techniques, not their value in optimization. Nonetheless, we did capture execution times, and found that they were reduced by 1-20%, but the degree of improvement was not well correlated with the dynamic removal counts. This information, combined with the knowledge that Marmot's checks are quite efficient, suggest that other effects, such as altered instruction schedules and register pressure changes, may be responsible for much of the improvement.

5.5 Analysis times

Table 2 gives relative analysis time costs for the various configurations of our analysis. For the most part, analysis times grew by less than 60%, but some outlying values did raise concerns:

– The additional cost of subtype expansion strongly depends on the representation used. Our prototype representation (see Section 3.2) factors attributes, but requires explicit instantiation of shared field representatives when a type restriction occurs. This is problematic for programs having many fields of reference type, and would need to be addressed in any serious implementation.
– For `javac` and `jess`, the cost of mutation tracking with fixed point iteration is significant. Given that relatively few call sites are visited multiple times during iteration, we suspect that much of the cost lies in searching callee-side signatures to determine if a unification operation is required.

Table 2. Relative analysis times (base=1)

name	s	m	mf	sm	smf
`spec_213_javac`	5.43	1.01	13.73	5.84	22.33
`spec_202_jess`	1.16	1.06	5.67	1.23	3.17
`spec_209_db`	0.96	1.11	1.37	1.02	1.00
`spec_228_jack`	0.38	1.04	1.58	0.39	0.47
`jlex100`	1.07	1.04	1.15	1.07	1.15
`javacup`	0.96	1.03	1.55	0.99	1.09

6 Related Work

6.1 Precision of equality-based analyses

Precision improvements for equality-based analyses can be broadly grouped into two categories. One group of techniques refines the granularity at which individual objects are modeled (e.g., by adding equality representatives for individual fields [Ste96a]). This finer granularity allows for a larger set of equivalence classes, some of which may refer only to some "aspects" of an object. Subtype expansion belongs to this category.

Another group of techniques relies on avoiding equalities where possible. Summary based context sensitivity does this at call sites, where equality in the callee need not induce equality in the caller. Mutation tracking is merely a way to avoid a specific class of such propagation. Other variants combine equality and flow constraints. Das's OLF technique[Das00] is of particular relevance in that it preserves flow directionality only for top-level interactions between objects, and requires equality for "content" properties such as pointer targets. Our techniques lack the precision of flow constraints, but do have the advantage of applying equally to all aspects of an object. Combining our techniques with OLF might yield interesting results.

6.2 Other context-sensitive analyses

A number of non-equality-based, context-sensitive flow analyses have been developed in recent years. These systems either construct and explicitly instantiate method summaries [CGS$^+$99,WR99,LH99,CRL99,Ruf00] to avoid pollution from multiple call sites or add instantiation constraints and flow polarities to enable context-sensitive queries after the initial analysis is complete [FRD00,RF01]. In both cases, adding subtype expansion should be straightforward, as it only requires changes to the object representation and not to the propagation of constraints. Mutation tracking, on the other hand, adds a new variety of intraprocedural information that guides constraint propagation at call sites, and thus might require additional modification.

7 Conclusion

We have presented two new, orthogonal techniques for improving the precision of equality-based dataflow analyses:

– **Subtype expansion** extends the static object representation to model equality between objects viewed at a particular shared subtype (or subtype ideal). This is similar to the extension of simple equality models with explicit representatives (types) for object fields, but operates in an orthogonal dimension. The extension of unification and context-sensitive call mapping to the expanded representation is straightforward, suggesting that systems using flow dependences could be similarly extended. Practical application of subtype extension will require additional experimentation with precision/complexity tradeoffs in the representation of subtype-specific information.
– **Mutation tracking** explicitly models the mutation of state reachable from formal parameters, avoiding a class of false aliases induced by the bidirectional nature of equality constraints. This technique applies primarily to summary-based context sensitivity techniques where the absence of mutation can be used to avoid the propagation of an alias or other dependency from a function to its caller(s). When combined with fixed point iteration, mutation tracking can provide a coarse-granularity form of flow directionality at recursive call sites.

Acknowledgements

We thank the ex-members of the Advanced Programming Languages group at Microsoft Research for developing the Marmot compiler infrastructure, and Todd Knoblock, Bjarne Steensgaard, and Manuvir Das for interesting and relevant conversations.

References

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, USA, 1986.

[CGS+99] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for Java. In *Proceedings of the 14th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '99)*, November 1999.

[CRL99] Ramkrishna Chatterjee, Barbara G. Rynder, and William A. Landi. Relevant context inference. In *Proceedings 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 133–146, January 1999.

[Das00] Manuvir Das. Unification-based pointer analysis with directional assignments. In *Proceedings of the SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 35–46, 2000.

[DGC98] Greg DeFouw, David Grove, and Craig Chambers. Fast interprocedural class analysis. In *Proceedings 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 222–236, 1998.

[FKR+00] Robert Fitzgerald, Todd B. Knoblock, Erik Ruf, Bjarne Steensgaard, and David Tarditi. Marmot: An optimizing compiler for Java. *Software: Practice and Experience*, 30(3):199–232, March 2000.

[FRD00] Manuel Fähndrich, Jakob Rehof, and Manuvir Das. Scalable context-sensitive flow analysis using instantiation constraints. In *Proceedings of the SIGPLAN 2000 Conference on Programming Language Design and Implementation*, June 2000.

[GDDC97] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call graph construction in object-oriented languages. In *Proceedings of the 12th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '97)*, pages 108–124, October 1997.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, June 1996.

[Hen91] Fritz Henglein. Efficient type inference for higher-order binding-time analysis. In *Functional Programming and Computer Architecture*, pages 448–472, 1991.

[Hen93] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, April 1993.

[LH99] D. Liang and M. J. Harrold. Efficient points-to analysis for whole-program analysis. In *Proceedings FSE'99*, pages 199–215. ACM, 1999.

[RF01] Jakob Rehof and Manuel Fähndrich. Type-based flow analysis: from polymorhphic subtyping to CFL-reachability. In *Proceedings 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 2001.

[Ruf00] Erik Ruf. Effective synchronization removal for Java. In *Proceedings of the SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 208–218, June 2000.

[Ste96a] Bjarne Steensgaard. Points-to analysis by type inference of programs with structures and unions. In *International Conference on Compiler Construction*, number 1060 in Lecture Notes in Computer Science, pages 136–150, Where?, April 1996.

[Ste96b] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 32–41, St. Petersburg Beach, FL, January 1996.

[Ste00] Bjarne Steensgaard. Thread-specific heaps for multi-threaded programs. In *Proceedings of the ISMM 2000 International Symposium on Memory Management*, July 2000. also published as SIGPLAN Notices 36(1), January 2001, ACM.

[WR99] John Whaley and Martin Rinard. Compositional pointer and escape analysis for Java programs. In *Proceedings of the 14th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '99)*, November 1999.

Appendix

Table 3. Raw benchmark results

(a) Static check operations

name	unopt	base	s	m	mf	sm	smf
spec_213_javac	321	262	212	261	261	212	197
spec_202_jess	95	49	37	49	48	37	37
spec_209_db	42	24	16	24	24	16	13
spec_228_jack	122	57	29	57	57	29	25
jlex100	67	36	34	34	34	34	34
javacup	434	384	60	384	384	60	53

(b) Dynamic check operations

name	unopt	base	s	m	mf	sm	smf
spec_213_javac	6408484	6308416	3476204	4799096	4799096	3476204	1910732
spec_202_jess	15071248	8666204	6485130	8666204	8666204	6485130	6485130
spec_209_db	53229831	19047	19047	19047	19047	19047	0
spec_228_jack	5703096	1086440	1028755	1086440	1086440	1028755	1002762
jlex100	113790000	61200	47000	47000	47000	47000	47000
javacup	1485504	1341350	598912	1341350	1341350	598912	589789

(c) Analysis times (sec)

name	base	s	m	mf	sm	smf
spec_213_javac	5.84	31.70	5.92	80.26	34.14	130.47
spec_202_jess	2.77	3.20	2.92	15.69	3.41	8.77
spec_209_db	0.88	0.84	0.97	1.20	0.89	0.88
spec_228_jack	4.56	1.72	4.75	7.22	1.78	2.16
jlex100	0.42	0.45	0.44	0.48	0.45	0.48
javacup	1.08	1.03	1.11	1.67	1.06	1.17

(d) Execution times (sec)

name	unopt	base	s	m	mf	sm	smf
spec_213_javac	2.843	2.843	2.780	2.796	2.828	2.812	2.812
spec_202_jess	2.202	2.171	2.139	2.171	2.171	2.155	2.186
spec_209_db	10.186	9.296	9.296	9.264	9.312	9.327	9.280
spec_228_jack	0.983	0.921	0.921	0.952	0.936	0.921	0.906
jlex100	3.781	3.014	3.046	3.124	3.093	2.999	3.015
javacup	0.189	0.186	0.184	0.183	0.186	0.180	0.180