Object representatives: a uniform abstraction for pointer information

Eric Bodden [1], Patrick Lam [2] and Laurie Hendren [1]

[1] School of Computer Science, McGill University
[2] Department of Electrical and Computer Engineering, University of Waterloo

eric.bodden@mail.mcgill.ca, *p.lam@ece.uwaterloo.ca*, *hendren@cs.mcgill.ca*

Abstract

Pointer analyses enable many subsequent program analyses and transformations by statically disambiguating references to the heap. However, different client analyses may have different sets of pointer analysis needs, and each must pick some pointer analysis along the cost/precision spectrum to meet those needs. Some analysis clients employ combinations of pointer analyses to obtain better precision with reduced analysis times. Our goal is to ease the task of developing client analyses by enabling composition and substitutability for pointer analyses. We therefore propose object representatives, which statically represent runtime objects. A representative encapsulates the notion of object identity, as observed through the representative's aliasing relations with other representatives. Object representatives enable pointer analysis clients to disambiguate references to the heap in a uniform yet flexible way. Representatives can be generated from many combinations of pointer analyses, and pointer analyses can be freely exchanged and combined without changing client code. We believe that the use of object representatives brings many software engineering benefits to compiler implementations because, at compile time, object representatives are Java objects. We discuss our motivating case for object representatives, namely, the development of an abstract interpreter for tracematches, a language feature for runtime monitoring. We explain one particular algorithm for computing object representatives which combines flow-sensitive intraprocedural must-alias and must-not-alias analyses with a flow-insensitive, context-sensitive whole-program points-to analysis. In our experience, client analysis implementations can almost directly substitute object representatives for runtime objects, simplifying the design and implementation of such analyses.

1. INTRODUCTION

Many static program analyses depend on the availability of pointer analysis information to disambiguate heap references. For instance, dependence and side-effect analyses need to know the identities of objects that are written to. An analysis that determines whether the fields of parameters are modified needs to know if such fields are modified through aliases. Typestate and tracematch analyses (as discussed in this paper) must acknowledge changes to object states even when objects are pointed to by many different variables.

While all of these client analyses need pointer information, they do not need the same precision in their pointer information. For instance, partial redundancy elimination, dead code elimination and structure copy optimization work adequately with imprecise pointer information [12]; other analyses, such as analyses for tracematches [3, 4] or for typestate [9], need more precise pointer information. Some approaches [4, 17, 33] combine multiple pointer analyses to get an adequate cost/precision trade-off.

However, it is currently difficult to combine multiple pointer analyses, as the interfaces to pointer analysis vary greatly: some analyses expose Boolean-valued methods which answer queries about local variables, while other analyses expose points-to sets which developers can test for disjointness. Furthermore, some pointer analyses return may-aliasing information, while others return must-aliasing information. Developers of client analyses must track the different pointer analysis interfaces and coordinate the calls to the different analyses, which complicates the design and implementation of these client analyses. We believe that a uniform interface for pointer analysis would greatly aid the development of client analyses.

We therefore propose *object representatives* as a convenient abstract interface that decouples pointer analyses from their clients. In this paper, we enumerate desirable properties of object representatives and propose ways of computing them. Object representatives assign names to heap objects and enable client analyses to uniformly 1) disambiguate references which definitely point to different heap objects (i.e. answer may-alias queries) and 2) identify references which definitely point to the same heap object (i.e. answer must-alias queries). We demonstrate that object representatives are an easy-to-use interface to pointer analysis that enables higher-level abstract reasoning about the heap and simplifies implementations of client analyses.

Object representatives can serve as static representatives of runtime objects; this works especially well due to the must-alias information that they encapsulate. In particular, representatives can be directly compared and indexed. They can therefore be stored in indexed data structures, such as hash sets or hash maps. The identity (and hash code) of an object representative is solely determined by its aliasing relationships with other representatives. In other words, two object representatives are equal if they must represent the same area of the heap. To sum up, object representatives provide client analyses with a way to reason about equality and inequality of references to the heap in a uniform, intuitive and well encapsulated way.

To concretely motivate the need for object representatives, we explain some details from our static analysis of tracematches [4], which requires precise flow-sensitive must-alias and must-not-alias information. Unfortunately, no currently-known algorithm can compute whole-program flow-sensitive must-alias information for sizable programs in a reasonable amount of time, so we combined three different pointer analyses. Our baseline pointer information comes from the Spark pointer analysis framework [21] and its extension, context-sensitive refinement-based whole-program points-to analysis [28]. Both are reasonably efficient, partly because they are not flow-sensitive—they do not consider the ordering of statements within a particular method—and partly because they do not support must-alias information. We then augmented this baseline information with intraprocedural flow-sensitive must-alias and must-not-alias information, which is also easy to compute efficiently. Object representatives allowed us to uniformly combine all of our analysis results and to hide them behind a single abstraction.

The contributions of this paper include:

- A description of object representatives and the properties that they should have.
- An algorithm for computing object representatives, which combines analysis results from different pointer analyses, such as interprocedural and intraprocedural may- and must-alias analyses.
- A discussion of our experience with object representatives, and in particular, how object representatives helped us design client analyses.

The structure of the remainder of this paper is as follows. Section 2 explains one particular client analysis that motivated our need for object representatives. Section 3 presents the abstract interface of object representatives, our notation for enabling different analyses to collaboratively name heap locations. In Section 4 we explain a reference implementation of this interface, tailored to the client analysis from Section 2. The implementation combines three different pointer analysis, described in Section 5, to disambiguate references to the heap. Finally, Section 6 presents related work and Section 7 concludes. The appendix contains additional information about more client analyses and how they can benefit from using object representatives.

2. CLIENT ANALYSIS: ABSTRACT INTERPRETATION OF TRACEMATCHES

Our need for object representatives was motivated by a number of problems that we encountered with traditional pointer abstractions when developing a particular client analysis. In previous work [3, 4] we proposed static analysis approaches for evaluating *tracematches* at compile time. Tracematches are a programming language feature for runtime monitoring. A tracematch associates a common typestate [9] with a number of objects using constraints. For instance, the constraint $i = o(\texttt{i2})$ on a state "called_next" could mean that `i2.next()` was just called on iterator object $o(\texttt{i2})$, referred to by program variable `i2` and identified by the abstract tracematch variable i. Such information is useful: programmers must never advance the iterator $o(\texttt{i2})$ (e.g. using a call to `i1.next()`, where

$o(\texttt{i1}) = o(\texttt{i2})$) without knowing that i has a next element (which can be checked by calling `i2.hasNext()`). We provide further information about tracematches in Appendix A.

To implement an abstract interpretation for tracematches, we needed to model runtime objects at compile time. That is, we had to represent a constraint $i = o(\texttt{i2})$ by a constraint $i = x$, where x is some compile-time representative of the runtime object $o(\texttt{i2})$. At first, it was not clear what this x should be.

2.1. Storing points-to sets.

In [3] we used points-to sets from a whole-program points-to analysis as our abstraction: we modelled the constraint $i = o(\texttt{i2})$ by $i = \texttt{pointsTo(i2)}$. To decide whether we had reached a fixed point or not, we needed to determine equality of two points-to sets, which was at best inelegant for points-to sets. Furthermore, we also found that the points-to sets were too imprecise for nontrivial tracematch analyses due to their flow-insensitivity.

2.2. Storing variable names.

Having found that whole-program points-to information was not strong enough on its own, we next sought (in [4]) to combine flow-sensitive intraprocedural must-alias and must-not-alias information with whole-program points-to information. We therefore needed a new compile-time representation for heap objects.

As a quick and dirty hack, we attempted to use local variable names, representing $i = o(\texttt{i2})$ with the constraint $i = \texttt{i2}$. After replacing all occurrences of points-to sets with local variables in our implementation (a tedious job!) we soon discovered the many disadvantages of storing variable names. For instance, we store constraints in disjunctive normal form (DNF), as hash sets of disjuncts, where each disjunct is stored as a hash map, mapping tracematch variable names to our abstraction. The use of local variable names with DNF caused unnecessary memory consumption. At runtime, if $o(\texttt{i1})$ and $o(\texttt{i2})$ point to the same object, then disjunct $i = o(\texttt{i1})$ equals disjunct $i = o(\texttt{i2})$. Therefore, the constraint only needs to store one of the disjuncts. At compile time, however, the disjuncts $i = \texttt{i1}$ and $i = \texttt{i2}$ would be considered different because `i1` and `i2` appear different. This would lead to both mappings being stored. The problem was that we conflated variable *names* with the names of the *values* stored in those variables.

Furthermore, it turned out that variable names are not enough. The same variable can point to different objects at different lines of the program (sometimes even at the same line, if this line is executed more than once). We would also have needed to store the source statements of local variables, which would have complicated the abstraction even further. This painful experience prompted our abstraction of runtime objects by object representatives.

Object representatives have applications beyond implementing abstract interpretation for tracematches. In Appendix B, we demonstrate how programmers of other client analyses can benefit from using object representatives by giving two concrete examples: constant propagation and side-effects analysis for method parameters.

3. OBJECT REPRESENTATIVES

Object representatives are inspired by work by Fink et al. on static evaluation of typestate [9]; in fact, object representatives are an extension to their notion of *instance keys* [9, 27]. Object representatives simplify the problem of statically reasoning about the heap by adding a layer of indirection. Namely, they enable analyses to reason about (static representations of) objects, rather than being forced to reason about variables.

Conceptually, object representatives must:

1. represent one or more runtime objects;
2. support a must-not-alias query that, when true, mean that object representatives i and j always refer to disjoint areas of the heap;

3. support an equality operation "`equals`" that, when true, mean that object representatives i and j always refer to the same heap object; `equals` must depend solely on a representative's must-alias relationship to other representatives.
4. belong to a certain must-aliasing scope.

The scope is necessary to concretize the general notion of must-aliasing. It enables the representative to say whether it must-aliases itself at a given location. One might think that variables must always alias themselves at a given location. This is not quite true in the presence of loops or recursion. Consider the example on the right. At line 5, `j` gets assigned a different object in every iteration of the outer loop. Depending on the client analysis, it can be valid both to answer that `j` does must-alias itself or that it does not. The answer depends on the scope: If the scope is the execution of the entire method `foo` then `j` not must-aliases `j` at line 7, due to the possible redefinition. If the scope is only the inner loop (lines 6-9), then `j` must-aliases itself at line 7: during the execution of the inner loop, the value of `j` never changes. Such a scope might be useful for analyses that attempt to determine loop invariants; other analyses may require method-wide, thread-wide or even program-wide scopes.

```
1 void foo( Iterator i) {
2   Collection c; Iterator j; Object o;
3   while(i.hasNext()) {
4     c = i.next();
5     j = c.iterator();
6     while (j.hasNext()) {
7       o = j.next();
8       /* use o somehow */
9     } } }
```

FIGURE 1: Example redefining `j` in the outer but not in the inner loop

Note that our abstraction will never say that two variables belonging to different scopes are must-aliased. For instance, if object representatives are defined on a method-wide scope, then we will use an intraprocedural must-alias analysis to relate these representatives to each other. Such an analysis cannot determine that two variables from different methods are must-aliased. To determine must-aliasing between these representatives, we must widen the scope, e.g. to a program-wide scope, and use an interprocedural must-alias analysis to compute object representatives.

We define two kinds of object representatives, *strong* and *weak* representatives. A strong representative represents at most one concrete runtime object during a single execution of the scope; weak representatives have no such requirement. In Section 5.1.3 we will explain how a must-alias analysis can generate both strong and weak representatives for a given scope. Client can select between generating strong and weak object representatives by setting a runtime flag; no changes to client code are needed.

Because object representatives implement `equals` in a sensible way, we can easily store them in associative data structures like hash maps. Since their identity is a function only of their relationship with other representatives, two representatives representing the same object but stored in different program variables will be equal. Object representatives therefore indeed directly *represent* runtime objects, and enable direct, intuitive and well encapsulated implementations of abstract interpreters.

Returning the example in Section 2, we now model the constraint $i = o(\mathtt{i2})$ by $i = r(\mathtt{i2})$, where $r(\mathtt{i2})$ is the object representative of the object that `i2` refers to. Here, the `equals` function identifies $r(\mathtt{i1})$ and $r(\mathtt{i2})$ as equal. Hence, we see that the constraints $i = r(\mathtt{i1})$ and $i = r(\mathtt{i2})$ are equal, which enables optimizations.

This concludes our abstract description of object representatives. In the next section, we explain one concrete implementation of object representatives, tailored to the static abstract interpretation of tracematches.

4. A CONCRETE IMPLEMENTATION

Figure 2 presents the abstract Java interface for object representatives. Note that object representatives provide must-alias and must-not-alias operations. Furthermore, they implement the generic method `equals` so as to satisfy the properties mentioned above. (The properties also apply to `hashCode`.) This interface would be easy to extend, e.g. with methods to retrieve the possible concrete types that a variable can be assigned. For the sake of brevity, we restrict ourselves to the interface given below.

The internal state of an object representative is defined in terms of three values, stored in fields of the `ObjectRepresentative` object:

1. a local variable name;
2. the statement for which the object representative is created; and
3. the representative's must-aliasing scope (e.g. loop, method, abstract thread, program, ...).

```
public interface ObjectRepresentative {
    public boolean mustAlias(ObjectRepresentative other);
    public boolean mustNotAlias(ObjectRepresentative other);
    public int hashCode();
    public boolean equals(Object obj);
}
```

FIGURE 2: Abstract interface for object representatives

Object representatives must contain a defining statement for the representative, since a variable name alone does not disambiguate potential objects in the presence of redefinitions. SSA form [1, 25] would mitigate this problem, but was not available to us; however, storing the defining statement was a perfectly acceptable solution in our situation.

We next sketch our implementation of the object representative interface, which is tailored to our particular context. The abstract interface of object representatives enables different implementations and different back-end analyses; we used a flow-insensitive refinement-based whole-program points-to analysis [28] with flow-sensitive intraprocedural must-alias and must-not-alias analyses.

4.1. Implementation of mustNotAlias.

The flow chart in Figure 3 illustrates the implementation of the `mustNotAlias(ObjectRepresentative)` method. First, we compare the two methods (i.e. scopes) of the receiver object representative and the parameter object representative. If the two object representatives come from the same method, we first apply our intraprocedural must-not-alias analysis (which is usually more precise). The analysis receives the associated local variables and assignment statements as input. If the intraprocedural must-not-alias analysis determines that those two variables at those statements cannot possibly point to the same object we return `true`, indicating that the object representatives cannot alias.

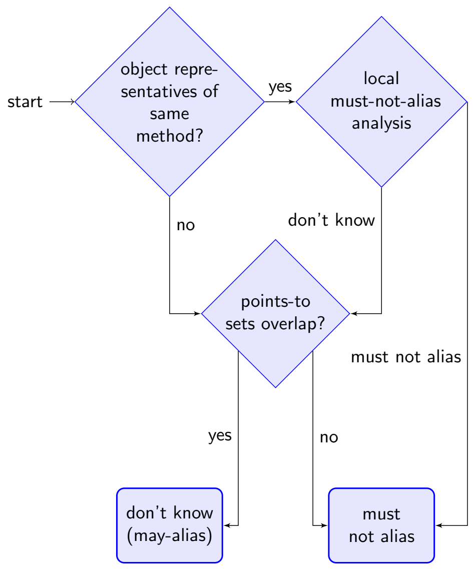

FIGURE 3: Flow chart for the must-not-alias operation on object representatives

If the intraprocedural must-not-alias analysis instead returns "don't know", or if the two variables come from different methods, we consult the interprocedural points-to analysis. Note that, sometimes, an interprocedural analysis can be more precise even if two variables are defined in the same method—for instance, they may be assigned fresh values in a callee method. We therefore construct points-to sets and refine them using context information for both object representatives. (Actually, we cache the refined points-to sets inside the object representatives). If the resulting points-to sets do not overlap, we return `true` as well, because again we know that the variables cannot be aliased. Otherwise, we return `false`, indicating that we don't know whether or not the variables may be aliased.

4.2. Implementation of mustAlias and equals.

The implementation of the method `mustAlias` is even simpler than that of `mustNotAlias`, because must-alias analysis cannot use our interprocedural points-to analysis. Only intraprocedural analysis information is available. We return `true` if the two object representatives belong to the same method and if the must-alias analysis for this method returns `true` on the local variables and statements stored in the object representatives. We implemented two different versions of the intraprocedural must-alias

```
public int hashCode() {
  final int prime = 31;
  int result = 1;
   result = prime * result + declaringMethod.fullName().hashCode();
   result = prime * result + must_alias_analysis (declaringMethod).valNumber(var,stmt);
  return result ;
}
```

FIGURE 4: Implementation of the `hashCode()` method for object representatives (pseudo code)

analysis, for strong and weak object representatives; clients may select the must-alias analysis that best suits their needs. Section 5.1 describes how we compute must-alias analysis results and how strong and weak analyses differ. The `equals` method simply delegates to the method `mustAlias` when its parameter object is an `ObjectRepresentative` as well.

4.3. Implementation of hashCode.

For performance reasons, hash codes should differ whenever possible; of course, they must be equal for two object representatives whenever an invocation of `equals` on those representatives would return `true`, i.e. if the representatives must be aliased. We found the following solution for computing an effective hash code for object representatives. The implementation combines (1) the identity of the declaring method; and (2) a unique value number assigned to the associated variable by the local must-alias analysis of this method at the associated statement (see Section 5.1). Figure 4 gives our concrete implementation in Java-like pseudo code. It ensures that hash codes differ whenever they can but are equal whenever they have to be.

5. POINTER ANALYSES

We integrated three different pointer analyses to compute our object representatives. Intraprocedural must-alias and must-not-alias analyses are quite precise in their area of applicability and run quickly, but do not have any information about object references passed in as method parameters or read from fields. We therefore incorporated a whole-program points-to analysis into our object representatives.

Our intraprocedural analyses can be seen as a generalized constant propagation over object reference values [31]. We generate fresh constant values either at expressions in general (must-alias) or at `new` expressions (must-not-alias). We then propagate values along assignments of local variables. The object representative abstraction enables client analyses to seamlessly combine the results of all three pointer analyses.

5.1. Intraprocedural must-alias analysis

Given two pointer variables `v1` at program point `p1` and `v2` at program point `p2`, our must-alias analysis determines whether `v1` and `v2` must point to the same heap object `o`. We assign value numbers to `v1` and `v2`; if they have the same number, then they must always point to the same heap object (for the duration of the method execution).

5.1.1. Value numbering.

Our local analyses assign a fixed value number [19] to each program expression in the code. (Identical expressions at different program locations are assigned different numbers.) If two variables contain the same value number, they must point to the same object: variables have the same value number when a variable is a copy of the value of another variable. Our abstract domain consists of integers and the special values $\top$ (unknown) and $\bot$ (nothing, for uninitialized values). Consider the following example.

The left-hand column shows value numbers before each line on the right. All variables initially get the uninitialized value $\bot$. Line 1 assigns the value number 1 to `i`. Line 2 copies 1 from `i` to `j`, so `j` and `i` are must-aliased. Our analysis indicates this by giving both `i` and `j` the same value number, 1, after line 2. After line 4, `i` gets a new value, modelled with value number 2.

```
(i,⊥), (j, ⊥)
(i, 1), (j, ⊥)    1 i = c1. iterator ();
(i, 1), (j, 1)    2 j = i;
(i, 1), (j, 1)    3 if (p) {
(i, 2), (j, 1)    4     i = c2. iterator ();
(i, 3), (j, 1)    5 } // 3 = { 1, 2 }
(i, 3), (j, 3)    6 j = i;
(i, 3), (j, 3)    7 print (j );
```

5.1.2. Value numbering at merge points.

If a variable (like `i`) has incoming value numbers 1 and 2 at a merge point, we generate a new value number "$\{1, 2\}$" to uniquely represent the two possible values of `i`, and create a unique identifier 3 to represent this value number. At line 6, we copy 3 to `j`, enabling us to conclude that `i` and `j` must be aliased. The creation of a fresh value number (e.g. 3) ensures that our analysis properly handles multiple reaching definitions; note that the fresh representative does not must-alias any of the previous reaching definitions (because we don't know which of the previous definitions actually reaches).

Values at different program points. Our must-alias analysis supports queries with respect to a statement. In the above example, `i` at line 1 and `j` at 6 are not must-aliased (due to the conditional). Our analysis can infer, however, that `i` at 1 and `j` at 2 are must-aliased, as `i` at 1 and `j` at 2 both have value number 1.

Queries with respect to statements are essential when client analyses need to associate analysis information with heap objects, not variables: a variable can point to many different objects over its lifetime. Static single assignment form (SSA form) [1, 25] can be used to encode the query position in the variable itself by splitting local variables at each redefinition. SSA form guarantees that there is a distinct variable name for any single *static* assignment, i.e. each assignment *location*. However, SSA form does not help handle variable redefinitions that are caused by multiple executions of the same statement.

5.1.3. Variable redefinitions in loops and reentrant code.

As noted in Section 3, each object representative is defined with respect to a single must-aliasing scope, e.g. a loop, method, thread or the entire program (other scopes may be possible, too). An object representative can be either strong or weak with respect to this scope. A strong representative only must-aliases itself at the same program location if it is guaranteed that this representative represents a *single* runtime object. The must-alias analysis presented above computes weak representatives—we therefore call it a *weak must-alias analysis*. It computes weak representatives because it assigns exactly one value number to each expression, even if the expression may be evaluated multiple times (by revisiting the same statement) and therefore could return different values (or objects) during the scope's execution. We found that weak representatives did not suffice for our tracematch analysis; referring back to Figure 1 on page 4, we needed to distinguish between the value held in variable `j` at different points in time. In particular, we had to determine whether the object pointed to by `j` had changed states. *Strong must-alias analysis* handles variable redefinitions by soundly over-approximating the variable contents. We can extend our weak must-alias analysis to a strong must-alias analysis as follows. We first find all strongly connected components (SCCs) in a method's control flow graph. If variable v gets value number i at an assignment statement within some SCC (i.e. within a loop), we invalidate i and give v the value number $\top$ as a conservative over-approximation.

Note that, in the setting of our concrete client analysis, variable redefinitions can only occur within loops: our scope for object representatives is a single method execution. Broader scopes would generally be helpful because they allow the resolution of must-alias queries from variables from different methods. However, they would also allow for other types of looping constructs, like recursion. (Such scopes could in principle be supported through other must-alias analyses. A global must-alias analysis could support a program-wide scope. However, we are not aware of any global must-alias analyses that are both practical and generally applicable.) Any such looping construct can cause a variable redefinition. Conversely, a smaller scope, e.g. a loop, might have fewer redefinitions and therefore require fewer representatives to be invalidated. If the scope contains no looping constructs (i.e. straight-line code), statements execute at most once, so a statement can never overwrite its definition from a previous execution. Invalidation is therefore unnecessary in straight-line code, and strong and weak representatives are equivalent. To sum up, we have seen that the definition of strong object representatives must depend on the scope for those representatives.

We present the strong analysis results for our example from Figure 1 on the right. Variables `v`, `j` and `o` are all

```
                                  void foo( Iterator i) {
                                      Collection c; Iterator j; Object o;
// (c,⊥), (i,1), (j,⊥), (o,⊥)         while(i.hasNext()) {
// (c,⊤), (i,1), (j,⊤), (o,⊤)             c = i.next();
// (c,⊤), (i,1), (j,⊤), (o,⊤)             j = c.iterator();
// (c,⊤), (i,1), (j,⊤), (o,⊤)             ...
// (c,⊤), (i,1), (j,⊤), (o,⊤)         }
                                  }
```

assigned ⊤ because they are reassigned in the loop. The variable `i`, however, is assigned the value number 1, because it is not assigned within the loop. Our analysis therefore answers that `i` always points to the same value during the execution of `foo()`.

Our implementation supports both strong and weak must-alias analyses. Our tracematch analyses use each analysis where it is most appropriate. For instance, we apply an optimization based on loop invariance. If the program's state does not change between loop iterations, then only the first iteration matters, so it is safe to use the weak must-alias analysis. Other optimizations require variables that may point to different objects at runtime to have different value numbers; we use the strong analysis for such optimizations.

5.2. Intraprocedural must-not-alias analysis

A must-not-alias analysis determines that two variables cannot point to the same heap object. If a must-not-alias analysis states that `v1` and `v2` may be aliased, then on any execution they may or may not be aliased. Conversely, if a must-not-alias analysis states that `v1` and `v2` cannot be aliased, then they *definitely* point to different objects, and we say that `v1` and `v2` "must-not-alias".

Must-not-alias information enables client analyses to conclude that two statements definitely do not interfere with each other because they act on different parts of the heap. We have implemented a must-not-alias analysis and include its results in our object representatives; like the must-alias information, it is based on value numbers.

The analysis abstraction for our must-not analysis consists of sets of value numbers, potentially including the initial value ⊥ (nothing aliases ⊥) and the unknown value ⊤ (everything aliases ⊤). Unlike the must-alias analysis, which assigns a fresh value number for each expression, the must-not analysis only assigns fresh value numbers at `new` expressions; otherwise it assigns ⊤. The results of other expressions are not necessarily fresh, and we cannot intraprocedurally say anything about the results of method calls, for instance; all such expressions may alias any other values in the method. Our analysis handles `x = y` by copying the value number(s) for `y` to `x`. For merge operations, instead of generating a new value number representing the join of both incoming value numbers as in the must-alias case, we simply combine the sets directly, making it easier to read off the must-not-alias relationships. Consider the following example.

```
                                  1  List l,m;
// {(l, ⊥), (m, ⊥)}               2  l = new List();          // 1
// {(l, { 1 }), (m, ⊥)}           3  m = new List();          // 2
// {(l, { 1 }), (m, { 2 })}       4  leak(m);
// {(l, { 1 }), (m, { 2 })}       5  if (p) {
// {(l, { 1 }), (m, { 2 })}       6      m = foo();
// {(l, { 1 }), (m, ⊤)}           7      l = new List();      // 3
// {(l, { 3 }), (m, ⊤)}           8  }
// {(l, {1,3}), (m, ⊤)}           9
```

At line 1, `l` and `m` are uninitialized, so we assign ⊥ to these variables. Lines 2 and 3 store fresh objects in `l` and `m`, so we give `l` and `m` new value numbers. At line 6, `foo()` might return a non-fresh object: for instance, `foo()` could return the object created at line 3, which was leaked in line 4. We must therefore assign ⊤ to `m`; it may alias anything. Line 7 stores another fresh object in `l`, so the analysis gives `l` the value number 3. After the merge point at line 8, `l` might contain either the object created at 1 or the object created at 3. For `m`, we merge $\{2\}$ with ⊤, giving an abstract value of ⊤ for `m`.

Querying whether `l` after line 3 could alias `l` after line 7 would give "must-not alias" because the set for `l` after line 3 is disjoint from the set for `l` after line 7. Conversely, `l` after line 3 and `l` after line 9 would result in the answer "may alias". A query of our abstraction on the relationship between `m` after line 4 and `m` after line 9 would also give "may alias".

5.3. Interprocedural points-to analysis

We have presented must-alias and must-not-alias analyses which work on one method at a time. Ideally, such flow-sensitive techniques would also work at a whole-program level, giving us detailed pointer information which integrates method parameters, array and field values. Unfortunately, no efficient techniques for a general interprocedural flow-sensitive must-alias analysis have yet been proposed. Flow-sensitive interprocedural approaches for the may-alias problem do exist [8, 18, 32], although even these approaches have not yet been shown to scale well enough to process interestingly-sized programs. However, flow-insensitive pointer analyses—points-to analyses—have been very successful.

Like our must-not-alias analysis, a points-to analysis determines the set of heap objects that program variables could possibly point to. Interprocedural points-to analyses assign value numbers to `new` expressions. (That is, `new` expressions continue to serve as representatives for memory locations.) However, interprocedural analyses can propagate value numbers throughout the entire program rather than through just a single method body, eliminating the need for conservative assumptions due to parameters, arrays and fields.

The example to the right illustrates the limitations of flow-insensitive analyses. Two different methods `f()` and `g()` assign to a field `x`. A main method then executes `f()` and `g()`, where `g()` invalidates the value of `x` set by `f()`. No flow-insensitive analysis could infer that `x` at line 7 is must-not-aliased with `x` after the assignment in `f()` in line 2. In other words, strong updates [5] are impossible. There would be just one points-to set for `x`, { ⟨1⟩, ⟨2⟩ }, modelling that `x` can point to the object created at (1) or the object created at (2).

```
1 A x;
2 void f() { x = new A(); }   // (1)
3 void g() { x = new A(); }   // (2)
4 void main() {
5     f();
6     g();
7     print (x);
8 }
```

Of course, interprocedural flow-insensitive points-to analyses can still be more precise than intraprocedural flow-sensitive analyses, even on two variables from the same method, because the interprocedural analysis can summarize computations outside the method that affect the method.

Consider the following example to the right. Here, no strictly intraprocedural analysis could have any information about the contents of iterator (), and must therefore assume that both calls to iterator () may, in principle, return the same value. We modelled this assumption in our intraprocedural analysis by assigning $\top$ as the value number for expressions that are not `new` expressions, such as method calls. Interprocedural analyses, on the other hand, typically consider the contents of called procedures, or at least summaries of their effects.

```
x = c. iterator (); // (3)
y = c. iterator (); // (4)
```

A context-sensitive interprocedural analysis can combine the body of the iterator function—which contains creation site (5)—with the fact that it is called from program points (3) and (4), to distinguish the values of x and y. A points-to analysis that uses calling-context information could model the object returned from iterator at (3) with a pair ⟨*call-site*, *creation-site*⟩, i.e. $\langle 3, 5\rangle$. The object returned at (4) would then be modelled with the pair $\langle 4, 5\rangle$. Because the points-to sets $\{\langle 3, 5\rangle\}$ and $\{\langle 4, 5\rangle\}$ are disjoint, the objects pointed to by x and y cannot alias. Interprocedural analyses give additional precision in some cases; we exploit this precision in the definition of object representatives.

```
public Iterator  iterator () {
    return new HashSetIterator(); // (5)
}
```

Context information—*where* is (5) called from?—is critical. Without context information, even an interprocedural analysis would model both x and y with just the creation site ⟨5⟩, making it impossible to conclude that x and y may not alias.

Applicability. It is difficult to soundly analyze partial programs when using a whole-program analysis, since the un-analyzed portions of these programs may have unknown and hard-to-circumscribe effects on the points-to information. Even if an interprocedural analysis can process any particular method quickly, the analysis generally needs to process all reachable methods to ensure soundness. It is therefore difficult to quickly obtain interprocedural analysis results; state-of-the-art machines require minutes to compute interprocedural analysis results. Our intraprocedural analyses, however, usually execute in milliseconds.

Demand-driven analyses. Our work on static tracematch optimizations uses Sridharan and Bodík's demand-driven and refinement-based context-sensitive points-to analysis [28], which is based on Spark [21], a context-insensitive flow-analysis framework, included in the Soot compiler framework [30]. We obtain analysis results by first running Spark to generate context-insensitive points-to sets, and then generating additional context information on demand. We only require context information for variables that we actually query. The demand-driven approach gives precise results (precise for a flow-insensitive approach, at least), while keeping analysis time to a minimum.

Availability. We integrated our implementation of object representatives into version 2.2.5 of the Soot program analysis framework (`http://www.sable.mcgill.ca/soot/`). Our static analysis of tracematches is available for download at `http://www.aspectbench.org/`, as well as a paper [4] describing the analysis in further detail. It is integrated as of version 1.3.0 of the AspectBench Compiler.

6. RELATED WORK

Past approaches to assigning names for heap objects include using `malloc` sites as names; Array SSA [10]; anchor handles [11]; and extended SSA numbering [19]. Most approaches explicitly assign names for heap objects by combining variable names with additional information. Our approach instead decentralizes the creation of names: conceptually, an object representative's identity is a function solely of its alias relationship to other representatives. Many must-alias and must-not-alias analyses exist in the program literature. All such analyses can be used to define object representatives.

We next discuss related work in the areas of SSA form, instance keys, whole-program pointer analyses, client analyses, flow-sensitive analyses, and specializing pointer analyses.

6.1. SSA form

Recall that the basic idea behind Static Single Assignment form [7] is to augment variable names with indices such that each augmented variable name has exactly one definition. Object representatives are more generally useful than SSA form: most importantly, they enable client analyses to smoothly integrate must-alias information. It is not clear how to relate must-alias information and SSA form. For instance, in the Array SSA work [10], the authors use ad-hoc techniques which combine SSA numbers with dominance information to get must-alias and may-alias information, and it is not clear how to fit interprocedural pointer analyses into this framework. Furthermore, the use of compiler-level objects, as in object representatives, helps analysis writers produce cleaner code.

6.2. Instance keys and memory disambiguation

Fink et al. put forward the notion of instance keys [9, 27], a term originating from Grove et al.'s InstVarKeys [13]. Fink et. al. informally define an instance key as an abstract name uniquely identifying a set of objects. While their work uses instance keys heavily, the authors do not describe the properties of instance keys nor how they are computed. We found that the notion of instance keys was useful in our own research, and it inspired our design of object representatives. To our understanding, instance keys also represent runtime objects and support must-not-alias queries. Object representatives add the notion of an identity, as defined using must-alias relationships, and the notion of a scope or lifetime over which this identity is defined.

Ghiya et al. [12] describe a memory disambiguation framework for the Intel Itanium compiler. Their work is similar to ours in that they provide an interface for static analysis clients to query a family of analyses (including intraprocedural and interprocedural pointer analyses) and expose abstract storage locations (LOCs). A LOC is defined to be a "a storage object that is independent of all other storage objects"; that is, a LOC never may-aliases any other LOC. There is no guarantee that two variables with the same LOC must alias each other. In contrast, object representatives do support must-alias information, and strong object representatives can therefore be used in the place of runtime objects when implementing static analyses.

In a similar vein, O'Callahan [23] proposes the Value-Point Relation. Unlike abstract storage locations, value points can describe any program expression. Two value points (v, ℓ_1) and (v, ℓ_2) are related if some execution of the program evaluates the same value for v at ℓ_1 and ℓ_2. Object representatives and value points both support queries with respect to program locations. However, like LOCs, the value-point relation is designed for *may* information, and it is not clear how to generalize the value-point relation to simultaneously support both may and must information, a key feature of object representatives.

Another way to combine must and must-not information is to compute them together using a combined abstraction. Emami et al. [8] propose such a combined abstraction and integrate it with a context-sensitive interprocedural pointer analysis, using the notation of abstract stack locations. Object representatives differ from abstract stack locations: object representatives give names for objects on the heap, while abstract stack locations track stack-based objects.

6.3. Whole-program pointer analyses

Our research relies on the existence of whole-program points-to analyses, which have been an active field of research over the last 25 years. Whole-program analyses generally trade off between precision and performance (in terms of time and space), and researchers have attempted to improve on precision while maintaining performance, or to maintain precision while improving performance.

In [15], Hind surveys the most important research in this field, lists 11 open questions in points-to analysis, and references papers that attempt to answer these questions. Open questions include: implementing points-to analyses for incomplete programs; designing demand-driven or incremental analyses; and efficiently incorporating flow-sensitivity at a whole-program level.

Two examples of whole-program pointer analyses are due to Altucher and Landi [2] and to Naik and Aiken [22]. Altucher and Landi name allocation sites and essentially use some limited context information by supporting custom allocation routines. They improve the results of their must-alias analysis with some may-alias information. Naik and Aiken propose a whole-program must-not-alias analysis which establishes facts of the form "if pointers p and q do not alias, then x and y do not alias either." Their analysis is a flow-insensitive whole-program approach which does not use must-alias information.

We have used whole-program points-to analyses to relate object representatives to each other. To answer aliasing queries on local variables, our component analyses only need to be able to determine whether the points-to sets for those variables overlap. We believe that all points-to analyses implement such an interface, so our approach is general enough to incorporate any whole-program pointer analysis. In fact, we experimented with a number of points-to analyses with different performance/precision trade-offs in the context of our static analysis of tracematches. Object representatives were flexible enough to incorporate all these analyses.

6.4. Client analyses

Many static analyses benefit from alias or points-to information, if they do not require it, since explicit references are a ubiquitous feature of modern programming languages. Pioli and Hind wrote a survey paper [16] which gives an overview of points-to analyses from a client's perspective and give examples of the wide range of extant client analyses. The authors mention live variable analysis, reaching definitions analysis and interprocedural constant propagation, and compare the precision of different points-to analyses for those client analyses. Their list is not exhaustive; other client analyses include cast elimination [28], side effect analysis [20], escape analysis [24] and ownership [6], thread-local objects analysis [14], static write barrier removal for generational garbage collectors [33], automatic parallelization and static race detection [22].

In general, all of the above client analyses work best with a maximally precise points-to or alias analysis, but they also generally require the whole program, triggering a need for a whole-program points-to analysis. However, intraprocedural flow-sensitive must-not-alias and must-alias information could enhance their precision. We believe that object representatives would help in the design and implementation of all of these analyses, as they did for our client analysis for tracematches.

An interesting client analysis that makes this fact apparent is the static analysis of PHP code for security vulnerabilities by Jovanovic et al. [17]. In their approach, the authors use intraprocedural (may- and must-alias) and interprocedural pointer analyses. The authors conflate heap objects with the variables that point to these objects, and maintain the relationships between variables manually, which leads to a complicated exposition.

6.5. Specializing pointer analyses

In [26], Rountev et al. present an analysis that treats different parts of a program with different precision. Many programs use large standard libraries that are both difficult and expensive to analyze precisely. The authors propose to analyze those libraries in a more coarse-grained fashion, computing only summary information for the libraries. This summary information can then enable a more precise analysis for the rest of the program. This work shares with our work the view that different pointer analyses might be suitable for different purposes. However, the approaches are orthogonal. Object representatives could be used to incorporate the results of analyses as proposed by Rountev et al. with each other and with the result of our analyses proposed here.

In [28] Sridharan and Bodík present a flow-insensitive, refinement-based, demand-driven, points-to analysis (which we used for our static tracematch optimizations). Their analysis supports the analysis of different parts of a program with different precisions, and enables client analyses to state how much they care about particular points-to sets. We believe that object representatives could support such an interface using parametrization: some object representatives could be computed more carefully than others, if they are particularly important to the client analysis.

7. CONCLUSIONS

In this paper we have described object representatives, a notation that assigns names to heap locations by combining the results of multiple pointer analyses. We presented an implementation of object representatives that combines results from a flow-insensitive interprocedural points-to analysis with results from flow-sensitive intraprocedural alias analyses. We demonstrated how we use object representatives computed this way in an abstract interpretation of tracematches, a programming language feature for runtime monitoring.

Object representatives enable clients to determine whether two variables definitely point to the same heap object and whether two variables definitely point to different objects. We furthermore defined two different kinds of object representatives, *strong* and *weak*, which have different semantics with respect to the redefinition of variables over the execution of a given scope.

We believe that object representatives simplify the design and implementation of static analyses that rely on pointer analyses: with object representatives, static analyses can be designed and implemented more like the runtime systems they model.

REFERENCES

[1] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '88)*, pages 1–11, New York, NY, USA, 1988. ACM Press.

[2] R. Altucher and W. Landi. An extended form of must alias analysis for dynamic allocation. In *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages (POPL '95)*, pages 74–84, January 1995.

[3] E. Bodden, L. J. Hendren, and O. Lhoták. A staged static program analysis to improve the performance of runtime monitoring. In E. Ernst, editor, *ECOOP 2007 - Object-Oriented Programming, 21st European Conference, Berlin, Germany, July 30 - August 3, 2007, Proceedings*, volume 4609 of *Lecture Notes in Computer Science*, pages 525–549. Springer, 2007.

[4] E. Bodden, P. Lam, and L. Hendren. Finding programming errors earlier by evaluating runtime monitors ahead-of-time. In *16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '08)*. ACM Press, Nov. 2008. To appear.

[5] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation (PLDI '90)*, pages 296–310, New York, NY, USA, 1990. ACM Press.

[6] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '98)*, pages 48–64, New York, NY, USA, 1998. ACM Press.

[7] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, Oct 1991.

[8] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation (PLDI '94)*, pages 242–256, New York, NY, USA, 1994. ACM Press.

[9] S. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. In *Proceedings of the 2006 international symposium on Software testing and analysis (ISSTA '06)*, pages 133–144, New York, NY, USA, 2006. ACM.

[10] S. J. Fink, K. Knobe, and V. Sarkar. Unified analysis of array and object references in strongly typed languages. In *Proceedings of the 7th International Symposium on Static Analysis (SAS '00)*, pages 155–174, London, UK, 2000. Springer-Verlag.

[11] R. Ghiya and L. J. Hendren. Putting pointer analysis to work. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '98)*, pages 121–133, New York, NY,

USA, 1998. ACM Press.

[12] R. Ghiya, D. Lavery, and D. Sehr. On the importance of points-to analysis and other memory disambiguation methods for C programs. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation (PLDI '01)*, pages 47–58, New York, NY, USA, 2001. ACM Press.

[13] D. Grove, G. DeFouw, J. Dean, and C. Chambers. Call graph construction in object-oriented languages. *SIGPLAN Notices*, 32(10):108–124, 1997.

[14] R. L. Halpert, C. J. F. Pickett, and C. Verbrugge. Component-based lock allocation. In *Proceedings of the 16th International Conference on Parallel Architectures and Compilation Techniques (PACT'07)*. IEEE, September 2007.

[15] M. Hind. Pointer analysis: haven't we solved this problem yet? In *Proceedings of the ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering (PASTE '01)*, pages 54–61, New York, NY, USA, 2001. ACM Press.

[16] M. Hind and A. Pioli. Which pointer analysis should I use? In *Proceedings of the ACM SIGSOFT international symposium on Software testing and analysis (ISSTA '00)*, pages 113–123, New York, NY, USA, 2000. ACM Press.

[17] N. Jovanovic, C. Kruegel, and E. Kirda. Precise alias analysis for syntactic detection of web application vulnerabilities. In *Proceedings of the ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS '06)*, 2006.

[18] W. Landi, B. G. Ryder, and S. Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation (PLDI '93)*, pages 56–67, New York, NY, USA, 1993. ACM Press.

[19] C. Lapkowski and L. J. Hendren. Extended SSA numbering: Introducing SSA properties to languages with multi-level pointers. In *International Conference on Compiler Construction (CC '98)*, volume 1383 of *LNCS*, pages 128–143. Springer, March 1998.

[20] A. Le, O. Lhoták, and L. Hendren. Using inter-procedural side-effect information in JIT optimizations. In R. Bodík, editor, *14th International Conference on Compiler Construction (CC '05)*, volume 3443 of *LNCS*, pages 287–304, Edinburgh, April 2005. Springer.

[21] O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. In G. Hedin, editor, *12th International Conference on Compiler Construction (CC '03)*, volume 2622 of *LNCS*, pages 153–169, Warsaw, Poland, Apr. 2003. Springer.

[22] M. Naik and A. Aiken. Conditional must not aliasing for static race detection. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '07)*, pages 327–338, New York, NY, USA, 2007. ACM.

[23] R. O'Callahan. The design of program analysis services. Technical Report CMU-CS-99-135, CMU School of Computer Science, June 1999.

[24] Y. G. Park and B. Goldberg. Escape analysis on lists. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation (PLDI '92)*, pages 116–127, New York, NY, USA, 1992. ACM Press.

[25] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '88)*, pages 12–27, New York, NY, USA, 1988. ACM Press.

[26] A. Rountev, S. Kagan, and T. Marlowe. Interprocedural dataflow analysis in the presence of large libraries. In *International Conference on Compiler Construction (CC '06)*, number 3923 in LNCS, pages 2–16. Springer, 2006.

[27] S. Shoham, E. Yahav, S. Fink, and M. Pistoia. Static specification mining using automata-based abstractions. In *Proceedings of the ACM SIGSOFT international symposium on Software testing and analysis (ISSTA '07)*, pages 174–184. ACM Press, July 2007.

[28] M. Sridharan and R. Bodík. Refinement-based context-sensitive points-to analysis for Java. In *Proceedings of the ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI '06)*, pages 387–400, New York, NY, USA, 2006. ACM Press.

[29] The AspectJ Team. The AspectJ Programming Guide. `http://www.eclipse.org/aspectj`.

[30] R. Vallée-Rai. Soot: A Java optimization framework. Master's thesis, McGill University, July 2000.

[31] C. Verbrugge, P. Co, and L. J. Hendren. Generalized constant propagation: A study in C. In *Proceedings of the 5th International Conference on Compiler Construction (CC '96)*, volume 1060 of *LNCS*, pages 74–90, April 1996.

[32] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA '99)*, pages 187–206, New York, NY, USA, 1999. ACM Press.

[33] K. Zee and M. Rinard. Write barrier removal by static analysis. In *Proceedings of the 17th Annual ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA '02)*, October 2002.

APPENDIX

In this appendix we give additional information about tracematches, the notation for runtime monitoring that motivated our need for object representatives. Also we explain how object representatives can benefit other client analyses, by giving the two examples of constant propagation and side-effects analysis for method parameters.

A. TRACEMATCHES

Tracematches are a programming language feature for runtime monitoring. A tracematch specifies a regular expression; the runtime system monitors for occurrences of the specified regular expression in the dynamic event stream of the program, and executes a given body of code when the regular expression matches. Our research aims to evaluate tracematches ahead of time, both 1) to generate better runtime monitoring code and 2) to verify programs for partial correctness, by proving that certain erroneous event sequences can never occur in program executions.

```
1  tracematch(Iterator i) {
2    sym hasNext before: call(* java.util.Iterator+.hasNext()) && target(i);
3    sym next before: call(* java.util.Iterator+.next()) && target(i);
4
5    next next { System.err.println("Trouble with "+i); } }
```

FIGURE 5: HasNext example: ensure `next` is not called twice without an intervening call to `hasNext`.

Figure 5 presents the `HasNext` verification tracematch, which matches (suspicious) traces where a program calls `i.next()` twice in a row without any intervening call to `i.hasNext()`. Tracematches include an alphabet of *symbols*, a *regular expression* over this alphabet and a *body* of code. Symbols define abstract tracematch events in terms of concrete program events, using AspectJ pointcuts [29]. Symbols may bind variables; line 1 of the tracematch declares that symbols in this tracematch may bind an `Iterator` object, `i`. Lines 2–3 define symbols `hasNext` and `next`, which capture method calls to the `hasNext()` and `next()` methods of `i`. Line 5 declares the regular expression "`next next`" (using symbols) and the body of code to be executed when the regular expression matches. The `hasNext` symbol on iterator `i` resets partial matches for `i`.

Now, assume that a programmer applies the `HasNext` tracematch to the example program in Figure 6. The program creates an iterator `i` and iterates on it. To demonstrate aliasing, we introduced auxiliary variables `i1` and `i2`. Aliasing is generally unavoidable. For instance, program optimizations and transformations can introduce aliasing, or variables enter or escape the current method and must be treated as potentially aliased.

```
1  Iterator i = c.iterator();
2  Iterator i1 = i;
3  while (i1.hasNext()) {          // hasNext(i1)
4      Iterator i2 = i;
5      Object o = i2.next();       // next(i2)
6  }
```

FIGURE 6: Example program using iterators.

We can observe that the program in Figure 6 is "safe" with respect to the tracematch—that is, the program can never trigger the tracematch. The key reason is that the variables `i1` and `i2` must point to the same object (they must-alias), so that each call to `next()` at line 5 must be preceded by a call to `hasNext()` on the same object, at line 3, so that there can never be two `next()` calls in a row without any intervening `hasNext()` calls.

A.1. Runtime evaluation

The runtime implementation of tracematches evaluates a tracematch by associating objects with automaton states. For each state, it records a logical formula, or constraint, which evaluates to true if an object is currently in a state; furthermore, it updates the formulas as the program executes. For the `HasNext` tracematch, the tracematch implementation uses the state machine from Figure 7, and assigns an initial configuration $(\mathbf{tt}, \mathbf{ff}, \mathbf{ff})$ to the state vector (q_0, q_1, q_2). This reflects the situation that, at program start, all objects reside in the initial state and in no other state.

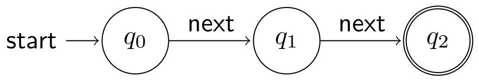
$$\begin{tikzcd}\text{start} \arrow[r] & q_0 \arrow[r, "\text{next}"] & q_1 \arrow[r, "\text{next}"] & q_2\end{tikzcd}$$

FIGURE 7: Automaton for the HasNext tracematch from Figure 5.

Once the program executes line 5 from the example (Figure 6), the `next` edge from q_0 to q_1 causes the runtime implementation to move the object stored in `i2` to state q_1. The new configuration is therefore $(\mathbf{tt}, i = o(\texttt{i2}), \mathbf{ff})$, where we denote the runtime object stored in `i2` with $o(\texttt{i2})$, and $i = o(\texttt{i2})$ at the second position represents the fact that $o(\texttt{i2})$ is now in state q_1. The constraint for q_0 remains $\mathbf{tt}$ because new partial matches can start any time; that is, a tracematch is matched against each suffix of the execution trace.

A.2. Abstract interpretation

At this point it should be clear that an abstract interpretation of tracematches somehow needs to abstract from concrete runtime configurations of the form $(\mathbf{tt}, i = o(\texttt{i2}), \mathbf{ff})$ by using abstract configurations. Object representatives allow us to simply use abstract configurations of the form $(\mathbf{tt}, i = r(\texttt{i2}), \mathbf{ff})$, where $r(\texttt{i2})$ is the object representative for the object stored in `i2`.

B. OTHER CLIENT ANALYSES

As we show in the following, object representatives can enable more simple and concise reasoning not only for abstract interpreters but also for other client analyses.

B.1. Constant propagation

Constant propagation is a basic compiler optimization that attempts to pre-compute values of expressions at compile time. Consider the following example program.

```
p.f = 1;
q.f = 2;
x = p.f + 1;
```

In the absence of pointer analysis, a client analysis does not know anything about the relationship between the values of variables p and q, and therefore could not conclude anything about the value of x after line 3.

Object representatives associate heap locations with variables. If two variables have the same strong object representative, then they definitely point to the same heap object. (We only use strong object representatives in the discussion in this section.) The implication is that constant propagation only needs to store one value per field per representative. If p and q have the same representative, then x gets 3 after line 3. When two variables have different object representatives, the client analysis can ask whether or not the object representatives are known to must-not-alias. If p and q's object representatives do must-not-alias, then x gets 2 after line 3.

B.2. Effects on method parameters

Developers would often like to know whether or not fields of method parameters are modified within the body of a method. The complication is that fields of method parameters can be written to indirectly, through aliases of the parameters, as seen in this simplified example:

```
1 void foo(T p) {
2     T q = p;
3     q.f++;
4 }
```

A sound analysis must be able to determine that p's field is modified by the write to q.f on line 3. If p and q carry the same strong object representative, then an analysis can state that p is definitely written to. Note how the object representative is usable in place of the actual runtime object. On the other hand, if p and q's object representatives must-not-alias, then the analysis can safely conclude that p is never written to in method foo.