Instance keys: A technique for sharpening whole-program pointer analyses with intraprocedural information

Sable Technical Report No. 2007-8

Eric Bodden, Patrick Lam and Laurie Hendren
School of Computer Science
McGill University
Montréal, Québec, Canada

October 20, 2007

Contents

List of Figures

Abstract

Pointer analyses enable many subsequent program analyses and transformations, since they enable compilers to statically disambiguate references to the heap. Extra precision enables pointer analysis clients to draw stronger conclusions about programs. Flow-sensitive pointer analyses are typically quite precise. Unfortunately, flow-sensitive pointer analyses are also often too expensive to run on whole programs. This paper therefore describes a technique which sharpens results from a whole-program flow-insensitive points-to analysis using two flow-sensitive intraprocedural analyses: a must-not-alias analysis and a must-alias analysis. The main technical idea is the notion of *instance keys*, which enable client analyses to disambiguate object references. We distinguish two kinds of keys: *weak* and *strong*. Strong instance keys guarantee that different keys represent different runtime objects, allowing instance keys to almost transparently substitute for runtime objects in static analyses. Weak instance keys may represent multiple runtime objects, but still enable disambiguation of compile-time values. We assign instance keys based on the results of our analyses. We found that the use of instance keys greatly simplified the design and implementation of subsequent analysis phases.

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. Typestate analyses must acknowledge changes to object states even if objects are pointed to by many different variables. An analysis that determines whether parameter fields are modified needs to know if such fields are modified through aliases.

While these analyses all need pointer information, they differ in the precision that they require. Some analyses, such as partial redundancy elimination, dead code elimination and structure copy optimization, appear to work adequately with imprecise pointer information [12], while other analyses, such as our own analysis for tracematches [3] or analyses for typestate [9], show a need for precise pointer information.

We believe that client analyses should be insulated from the internal workings of the pointer analysis. When designing a client analysis, it is tempting to conflate local pointer variables `v` with heap objects o. Such an approach might appear to sidestep pointer analysis questions. However, the shortcomings of such an approach quickly become obvious; client analysis implementers end up integrating pointer analysis into their own static analyses, hurting both the performance and maintainability of their own analyses. Modularity in analysis design enables improvements in pointer analysis to immediately benefit client analyses.

Our goal is to provide an interface to pointer analysis which enables client analyses to cleanly extract the information that they need. Hiding the details of pointer analysis can simplify the design and implementation of client analyses. Fink et al. have mentioned [9, 26] that *instance keys* can serve as a convenient interface to pointer analysis. In this paper, we explore the notion of instance keys and describe some of their properties. Essentially, instance keys assign names to heap objects and enable client analyses to 1) disambiguate references which definitely point to different heap objects and 2) identify references which definitely point to the same heap object.

There have been many approaches to assigning names for heap objects in the past. These include using `malloc` sites as names; Heap SSA [10]; anchor handles [11]; and extended SSA numbering [18]. Most approaches explicitly assign names for heap objects by combining variable names with additional information. Our approach differs from those approaches by decentralizing the creation of names: an instance key's identity is a function solely of its must-alias and must-not-alias relationships to other keys. We define and use generic interfaces to the backing must-alias and must-not-alias analyses, enabling the use of more sophisticated or specialized pointer analyses in the future. Our approach supports the modular, simultaneous development of pointer and client analyses.

This is particularly important due to the well-known trade-off between analysis precision and scalability. Because pointer analysis is inherently complex, an interprocedural analysis must sacrifice some precision if it is to complete in a reasonable amount of time. For instance, the Spark pointer analysis framework [21] and its extension, refinement-based points-to analysis [27], are reasonably efficient. However, they are not flow-sensitive—they do not consider the ordering of statements within a particular method. They also do not support must-alias information. On the other hand, intraprocedural approaches can easily support flow-sensitivity and must-alias information and still remain scalable, but lack precision if the values of variables from different methods have to be considered.

We found that our static analysis of tracematches benefited from information from intraprocedural pointer analyses [4]. Instance keys enabled us to combine the interprocedural information with analysis results from precise intraprocedural points-to analyses. Some sub-analyses exploited the defining property of *strong* instance keys. Different strong instance keys represent different runtime objects. This property enables the design and implementation of static analyses that resemble their dynamic counterparts: static instance keys can act as representatives for runtime objects.

We have implemented a static whole-program optimization using instance keys. We felt that it was significantly easier to implement this system using instance keys compared to our initial attempt, which stored not instance keys but variable names and then used must-alias, must-not-alias and points-to analysis results to resolve aliasing directly.

The contributions of this paper are:

- A description of instance keys and the properties that they should have.
- An algorithm for computing instance keys, which combines analysis results from intraprocedural and interprocedural pointer analysis.
- A discussion of our experience with instance keys and how they helped us design client analyses.

The structure of the remainder of this paper is as follows. Section 2 explains the context of our work and presents the pointer analysis questions that our notation can answer. Section 3 describes local pointer analyses that disambiguate some variables. Section 4 summarizes the whole-program points-to analysis that handles any remaining queries. Section 5 presents instance keys, our notation for enabling different analyses to collaboratively name heap locations. Finally, Section 6 presents related work and we conclude in Section 7.

2 Motivation

In this section we present a number of pointer analysis clients and explain how each of them uses pointer analysis. We present constant propagation; an analysis that determines if fields of object parameters are ever modified; and typestate.

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.

```
1 p.f = 1;
2 q.f = 2;
3 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.

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

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 instance key, then an analysis can state that p is definitely written to. Note how the instance key is usable in place of the actual runtime object. On the other hand, if p and q's instance keys must-not-alias, then the analysis can safely conclude that p is never written to in method foo.

Typestate, tracematches and bug finding. Typestate properties augment the type of a heap object with a time-variant state. Certain actions are not allowed on objects in certain states. As an example, consider the safety property that programs should not read from files that have been closed, and the following example method:

```
1 boolean compareone(Reader r1, Reader r2) {
2     int i = r1.read();
3     r1.close();
4     int j = r2.read();
5     r2.close();
6     return i == j;
7 }
```

This method reads a byte from r1 and r2 and compares them. In general, this method only satisfies the safety property if r1 and r2's instance keys must-not-alias, and the method definitely violates the property if r1 and r2 have the same instance key. Note that analyzing this method requires interprocedural information.

Now consider the property that readers should always be closed after they are opened, with the following code:

```
Reader r = new Reader();   // open the Reader
// ...
Reader r2 = r;
r2.close();
```

The code only satisfies the property if r and r2 point to the same heap object. If r and r2 have the same instance key, a static analysis can safely report that the property holds.

We return to the property that reads must occur on open files. Consider the following code:

```
1 while (...) {
2     Reader r = new Reader();
3     r.read();
4     r.close();
5 }
```

Note that different iterations of the loop have different r objects, which complicates pointer analysis. However, in each single iteration, the instance key for r at line 3 will be in the state "open", enabling an analysis to verify that the call to read satisfies the safety property. In this work, we define two different kinds of instance keys, *weak* and *strong*. Strong keys model must-alias relationships soundly even over multiple loop iterations.

We have recently investigated tracematches [3,4], which subsume typestates by enabling developers to state and verify properties of *multiple* correlated objects. The static analysis of tracematches and related tools for bug finding and program verification all require pointer analysis, if they attempt to reason at all about properties of heap objects at compile time. Our tracematch analysis uses both weak and strong instance keys.

2.1 Naming heap locations

At compile time, the program under analysis manipulates reference-typed local variables. Local variables are far more difficult to work with than heap objects: it can be difficult to determine whether local variables v_1 and v_2 point to the same object. Our goal was to find a notation which would enable the design of static analysis algorithms that are as similar as possible to dynamic analyses, and we believe that instance keys fill this niche: if two object references have the same instance key, then at runtime the references must point to the same object. The design of instance keys enables our static analysis to answer the following two questions:

1. Given object references v_1 and v_2 at program points p_1 and p_2, must v_1 and v_2 point to different heap objects? (must-not-alias analysis)

2. Given object references v_1 and v_2 at program points p_1 and p_2, must v_1 and v_2 point to the same heap object? (must-alias analysis)

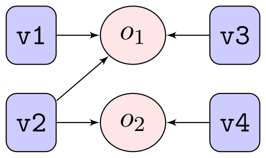

Figure 1: v_1, v_2, v_3, v_4 local variables; o_1, o_2 heap objects. Variables v_1, v_2 may-aliased; v_1, v_3 must-aliased; v_1, v_4 must-not-aliased.

Figure 1 illustrates the situations that we are interested in. The figure contains ellipses with heap objects o_1, o_2 and local variables `v1` through `v4`. We do not know whether `v2` points to o_1 or o_2. The query is whether two local variables must alias or must not alias; we know that `v1` and `v3` must-alias, while `v1` and `v4` are must-not-aliased. We have no precise information about `v2`: it does not must-alias nor must-not-alias any other variable.

Note that we generalize the typical setting of pointer analysis by specifying the program points where we are querying the objects in addition to just the local variables. We need to specify program points because we are comparing the value contained in reference `v1` at point `p1` with the value in `v2` at point `p2`; between `p1` and `p2`, both `v1` and `v2` could change values. (This is even true if `p1`=`p2`, when `p1` is inside a loop.) Instance keys encapsulate information about the variable and the program point where we are asking about that variable.

We have been careful to distinguish between the pointer variable `v` and the heap object o. When reasoning about programs, it is tempting to conflate variables with their contents; however, we found it impossible to develop precise static analyses for tracematches without carefully distinguishing variables and heap objects. The notion of instance keys adds a layer of indirection and enables us to design our analysis around a stable representation of heap objects.

Pointer analysis is broadly applicable as a client analysis for a range of static analyses. We therefore believe that instance keys will be useful in many situations.

3 Intraprocedural Pointer Analyses

We first attempted to answer our questions about object references using intraprocedural techniques. Such techniques are quite precise in their area of applicability and run quite quickly. Unfortunately, intraprocedural analyses do not have any information about object references passed in as method parameters or read from fields, and we found that we had to combine our intraprocedural analyses with the whole-program points-to analysis described in Section 4.

Our intraprocedural analyses can be cast as a generalized constant propagation over object reference values [30]. Depending on whether we wish to decide whether two variables must or must-not alias, fresh constant values are either generated at expressions in general (must-alias) or at `new` expressions (must-not alias). Values are then propagated along assignments of local variables.

3.1 Must-alias analysis

We next present our 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 they must point to the same heap object `o`. Our analysis solves this problem by assigning value numbers to variables. We designed the analysis so that if it assigns `v1` and `v2` the same value number, then they must always point to the same heap object on all program executions.

Note that if two variables have the same number, then they also *may* alias each other. Conversely, if two variables must-not alias, then they will be represented by different instance keys. We will return to this point when we discuss instance keys in Section 5.

3.1.1 Value numbering

We now explain the value numbering approach in more detail. We base our solution on global value numbering and extended SSA numbers [18]. Our intraprocedural must-alias analysis uses value numbering to identify and discriminate different heap locations. We assign a fixed value number to each program expression. (Note that identical expressions at different program locations are assigned *different* numbers.) We maintain the invariant that if two variables contain the same value number, then 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 value numbers and the special values ⊤ (unknown) and ⊥ (nothing, for uninitialized values). Consider the following example:

```
 1  // {(i,⊥), (j, ⊥)}
 2  i = c1.iterator();      // (1)
 3  // {(i,(1)), (j, ⊥)}
 4  j = i;
 5  // {(i,(1)), (j,(1))}
 6  if (p) {
 7      i = c2.iterator(); // (2)
 8      // {(i,(2)), (j,(1))}
 9  }
10  // {(i,⊤), (j,(1))}
11  j = i;
12  // {(i,⊤), (j,⊤)}
13  print(j);
```

At the beginning all variables are assigned the uninitialized value ⊥. Line 2 then assigns the value number (1) to `i`. At line 4, the value (1) is copied from `i` to `j`, so `j` now points to the value that `i` is also pointing to, i.e. `j` and `i` are must-aliased. Our analysis models this by propagating the number (1) to `j` when processing the assignment at line 4.

After line 7, things look different, since `i` is assigned a new value. To model this situation, we assign `i` the value number (2).

When computing the analysis information before line 10, we must merge values (1) and (2) for `i`. A conservative approximation yields the unknown number, ⊤. A value of ⊤ does not must-alias any other value, not even itself.

One problem is now that, once `i` is assigned $\top$, the analysis no longer has any must-alias information for `i` (until it gets redefined). In line 11, when `j` is copied from `i` again, both variables point to $\top$. Because $\top$ does not must-alias any other value, we have to assume that `j` does not must-alias `i`, even though after line 11 it clearly does. This imprecision is unnecessary, and we next describe how to avoid this imprecision.

3.1.2 Value numbering at merge points

At merge points, if two different value numbers for a variable `v` are available, the above algorithm erases all information about `v`, because it is overly conservative and simply stores $\top$ as the value for `v`, which does not give us any exploitable information.

But if the variable `v` has incoming value numbers (1) and (2) at the merge point, then we can generate the value number $\{(1),(2)\}$ to uniquely represent the two possible values of `v`. Hence, before line 10 in the example, we can merge the information $\{(i,(1)),(i,(2))\}$ to $\{(i,(3))\}$, where (3) is a unique identifier for the set $\{(1),(2)\}$. At line 11 we copy (3) to `j`, enabling us to conclude that `i` and `j` must be aliased.

Values at different program points. Note that our must-alias analysis can be queried *with respect to a statement*. In the above example, `i` at line 1 and `j` at 10 are not must-aliased (due to the conditional). However, our analysis can infer that `i` at 1 and `j` at 4 are must-aliased: `i` at 1 and `j` at 4 both have the 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) [1,24] can be used to encode the query position in the variable itself by splitting local variables at each redefinition.

3.1.3 Treatment of redefinitions in loops

SSA form guarantees that there is a distinct variable name for any single *static* assignment, i.e. each assignment *location*. Thus, naively one might think that variables must always alias themselves, because they are only assigned at a single point. We experienced a situation where this was not quite true. Consider the following example:

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

At line 5 in the above example, `j` is assigned a potentially different object in every iteration of the outer loop. Hence, in such a situation, depending on the client analysis, there might be two possible answers to the question whether `j` must alias itself at line 5. The analysis that we stated above would conclude that `j` must alias itself, because for the same expression at the same program location we assign the same fixed value number. For some client analyses this might be a sound and precise result, and we provide examples of such analyses below. However, some analyses can take into account that `j` can indeed point to different values over its lifetime at the same line number 5. One of our static analysis for tracematches is such a case. Because tracematches can reason about the use of values *over time*, we had to distinguish between values the same variable was assigned earlier from the value it has at the current time of execution. For this purpose we designed a second variant, *strong must-alias analysis*, that handles redefinitions in loops via a sound over-approximation. By contrast, we call the must-alias analysis we defined above a *weak must-alias analysis*.

The definition of the strong must-alias analysis differs only slightly from its weak counterpart. In the weak approach we assigned a *fixed* value number to each expression. Once this value was assigned, it would never change (at that location). For the strong analysis, we instead do the following. If we visit a statement of the form `v = exp` for the first time and `exp` is not a local variable, then we assign `v` a fresh value number. If we then visit the statement a second time, implying that the statement belongs to a loop, then we assign `v` the conservative value $\top$. That way, values that are redefined within loops will ultimately be assigned $\top$, a sound over-approximation. In our example, the strong analysis results would look like follows.

```
void foo(Iterator i) {
    Collection c; Iterator i, j; Object o;
    // {(c,⊥), (i,⊥), (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,⊤)}
}
```

Note that `v`, `j` and `o` are all assigned $\top$ because they are reassigned. The variable `i`, however, is assigned the value number (1), because it is not visited twice in the computation of the analysis result. Hence, one can correctly infer that `i` will always point to the same value during the execution of the method.

Our current implementation supports both kinds of must-alias analysis. Our intraprocedural client analysis for tracematches uses each kind of analysis where it is most appropriate. For instance, we apply one optimization based on loop-invariance. Tracematches happen to have the nice property that, if their evaluation is invariant in the first loop iteration, then their evaluation is also invariant for subsequent iterations. A static analysis may therefore consider only the first iteration. In such a situation, it is sound to use the weak must-alias analysis. Other optimizations we designed are only sound if variables that may point to different objects at runtime do not have the same value number (or at least one of them has $\top$), even if this concerns only one variable redefined in a loop. For such optimizations we use the strong must-alias analysis.

3.2 Must-not-alias analysis

A must-not-alias analysis determines whether 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. In such a case, we say that `v1` and `v2` "must-not-alias".

In Section 2 we presented a number of applications of must-not-alias information. In brief, 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; 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, the initial value $\bot$ and the unknown value $\top$ (representing the full set that contains everything). 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 $\top$. This is because for other expressions, e.g. method calls, one cannot intraprocedurally guarantee that a fresh value is computed. If the value is not fresh, it may be aliased. Our analysis handles `x = y` by copying the value number 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 join the sets directly. This makes it easier to read off the must-not-alias relationship in the end. Consider the following example.

```
1  List l,m;
2  // {(l, ⊥), (m, ⊥)}
3  l = new List();              // (1)
4  // {(l, {1}), (m, ⊥)}
5  m = new List();              // (2)
6  // {(l, {1}), (m, {2})}
7  leak(m);
8  // {(l, {1}), (m, {2})}
9  if(p) {
10     // {(l, {1}), (m, {2})}
11     m = foo();
12     // {(l, {1}), (m, ⊤)}
13     l = new List();          // (3)
14     // {(l, {3}), (m, ⊤)}
15 }
16 // {(l, {1,3}), (m, ⊤)}}
```

In line 2, the values of `l` and `m` are not yet known, so we assign $\bot$ to these variables. Line 3 stores a fresh object in `l`, so we update it with a new value number. In line 5, the same happens with `m`. In line 11, we cannot be sure whether or not `foo()` returns a fresh object: for instance, `foo()` could return the object created at line 5, which was leaked in line 7. Hence, we assign $\top$ to `m`, representing "anything". In line 13, another fresh object is stored in `l`, and the analysis hence gives `l` the new value number (3). At the merge point in line 16, `l` might contain either (1) or (3): we know that it contains either the object created at (1) or the object created at (3). For `m`, we merge $\{2\}$ with $\top$, giving an abstract value of $\top$ for `m`.

Recall that our analysis can say whether variables must-not-alias other variables at given program points. Querying whether `l` at line 4 could alias `l` at line 14 would give "must-not alias" because the set for `l` at line 4 is disjoint from the set for `l` at line 14. Conversely, `l` at line 4 and `l` at line 16 would result in the answer “may alias”. A query on the relationship between `m` in line 6 and `m` in line 16 would also result in “may alias”, which is sound: at line 16, the call to `foo()` might have executed and returned the object created at line 5.

4 Interprocedural points-to analysis

In the previous section, we presented must-alias and must-not-alias analyses that work on a single method at a time. Ideally, such flow-sensitive techniques would also work on a whole-program level, giving us even more detailed pointer information (since method parameters and field values could be known). 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 [7, 17, 31], although even these approaches have not yet been shown to scale well enough to process interestingly-sized programs. Flow-insensitive approaches to the pointer analysis problem, however, have been very successful. Most of these approaches are classified as points-to analyses.

Like our must-not-alias analysis, a points-to analysis determines the set of values (heap objects) that reference-typed program variables could possibly point to. Interprocedural points-to analyses resemble our local analyses in that they assign value numbers to `new` expressions. (In other words, `new` expressions continue to serve as representatives for memory locations, possibly augmented with context information, as we explain below.) 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 and fields.

The following example illustrates the limitations of flow-insensitive analyses.

```
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 }
```

Two different methods `f()` and `g()` assign to a field `x`. A main method then executes `f()` and `g()`, with the execution of `g()` invalidating 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 single points-to set for `x`, $\{(1), (2)\}$, modelling that `x` can point to the object created at (1) or the object created at (2).

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:

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

Here, any intraprocedural analysis would have no information about the internals of iterator (), and must therefore assume that both calls to iterator () may, in principle, return the same value. We modelled this assumption in our local 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 use the internals of called procedures, or at least summaries of their effects. The iterator function looks something like this:

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

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 of the form (*call-site*, *creation-site*), i.e. $(3, 5)$. The object returned at (4) would then be modelled with the pair $(4, 5)$. Because the points-to sets $\{(3, 5)\}$ and $\{(4, 5)\}$ are disjoint, the objects pointed to by x and y cannot alias. Observe that interprocedural analyses give additional precision in some cases; we will exploit this precision in the definition of instance keys.

Context information—*where* is (5) called from?—is critical in the above example. 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. With a whole-program analysis, it is difficult to soundly analyze *partial* programs, since the unanalyzed portions of these programs may have unknown and hard-to-circumscribe effects on the points-to information. Another concern is efficiency: Even if an interprocedural analysis can process any particular method quickly, the interprocedural analysis generally needs to process all reachable methods to ensure soundness. It is therefore difficult to quickly obtain interprocedural analysis results; in particular, state-of-the-art machines require minutes to compute interprocedural analysis results. Our local analyses, on the other hand, usually execute in milliseconds.

Demand-driven analyses. In our work on static tracematch optimizations, we used Sridharan and Bodík's demand-driven and refinement-based context-sensitive points-to analysis [27]. It is based on Spark [21], a context-insensitive flow-analysis framework which is included in the Soot compiler framework [28]. 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 (as precise as possible for a flow-insensitive approach, at least), while keeping analysis time to a minimum.

5 Instance keys

"There are no silver bullets." We have seen that different situations call for different pointer analyses. In general, flow-sensitive analyses are most precise, but such analyses only work intraprocedurally. Any values from outside a method must be conservatively approximated. Interprocedural flow-insensitive approaches eliminate the need for conservative approximations in some cases, but they cannot use information that is encoded in the ordering of program statements. We would like to combine the strengths of the intraprocedural and interprocedural approaches: given a pointer analysis query on two variables, we would like to call the optimal sequence of component analyses to answer our query.

An additional complication is that the different component analyses have different interfaces. For instance, a points-to analysis returns points-to sets for a given variable. Points-to sets can be tested for non-empty intersection ("overlap") with other points-to sets. If the points-to sets for two variables do not overlap, then the variables cannot alias. On the other hand, the intraprocedural must-alias and must-not-alias analyses answer the pointer analysis question directly: these analyses take variables as input and return a boolean value (must-not-aliased or don't know; must-aliased or don't know) as output. However, the intraprocedural analyses are flow-sensitive and therefore take an additional input: they need to know the program points that the client analysis is interested in.

We attempted to design and implement a client analysis (for static tracematch optimization) that directly queried these intraprocedural and interprocedural pointer analyses. We found that a direct integration of these different analyses carried significant conceptual and implementation-level overhead. We fought code duplication and scattering. Because our flow equations used variables, we had to litter our code with `if-then-else` conditionals to select appropriate analyses for appropriate variables. Furthermore, we had to update flow sets when variables were updated, directly and indirectly. Our code did not look much like its dynamic equivalent.

Conceptually, the problem was that our static analysis conflated objects with the variables that point to those objects. If the code contained an assign statement `p = new O();`, we represented the newly created object with the variable `p`. Variables are poor representatives for objects. One problem was that multiple variables can point to the same object, which made it difficult to store analysis information. Consider the following snippet of code.

```
1 Iterator i = new Iterator();
2 if(i.hasNext()) {
3     j = i;
4     j.next();
5 }
```

We found that our analysis was quite difficult to implement properly in the presence of such code. Consider variable `i`. At line 2, we could store information about `i` by associating it with `i`, and we could try to store the information in a hash map, indexing the information by `i`. Now, line 4 operates on variable `j`. But `j` and `i` represent the same object, and the information is indexed on `i`. How could we look up this information? After all, line 4 makes no mention of the name "`i`". So we would have to iterate through all possible variable bindings, see if any of them must-alias `j`, and if so, look up the value associated with that binding. Such an approach leads to both poor performance and unreadable code.

Our solution, instance keys, was inspired by work by Fink et al. on static evaluation of typestate [9]. Instance keys simplify the problem by adding a level of indirection. Conceptually, they enable analyses to reason about (static representations of) objects, not variables.

5.1 Properties of instance keys

The following properties should hold for instance keys:

1. Instance keys represent one or more runtime objects.
2. Instance keys support a must-not-alias operation that answers the question of whether two keys can never actually represent the same object.
3. Two *strong* instance keys are equal only if they are known to represent *the same* concrete object. *Weak* instance keys are equal only if they represent concrete objects created at the same statement.

These properties enable direct and natural implementations of static abstract interpretations of programs. Whenever at runtime a concrete object would be assigned, we can "assign" to the instance key at compile time. Because two instance keys are equal if the objects they represent are the same, we can store instance keys in associative data structures, such as hash tables.

In the example that gave us such trouble when using variables as representatives for heap objects, we instead store the instance key of `i` at line 1 instead `i`. For the look-up at line 3, we can use the binding for the instance key of `j`, which will be equal to the instance key for `i` and thus have the same hash code.

5.2 Implementation of instance keys

Figure 2 presents the abstract Java interface for instance keys. They implement a must-alias and must-not-alias operation. Furthermore, they implement the generic methods `equals` and `hashCode` in a way that satisfies the above mentioned properties.

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

Figure 2: Abstract interface for instance keys

The internal state of an instance key is defined via three values, stored in fields of the `InstanceKey` object:

1. a local variable name;
2. the statement for which the instance key is created;
3. the method in which the local variable is defined (and hence the method in which the assignment statement resides).

Those three parameters are passed to the constructor of each instance key. The statement parameter is necessary for the reasons explained earlier: The same variable in the same method can generally be assigned multiple values in different statements. Note that this could be worked around by using SSA form [1, 24]. When using SSA form, the statement is encoded in the variable name. For technical reasons, using SSA form was not an option in our client analysis, so we decided to store the defining statement explicitly.

5.2.1 Implementing the method `mustNotAlias`

Based on the state of an instance key, we can implement the method `mustNotAlias(InstanceKey)` as shown in the decision diagram in Figure 3. First, we compare the two methods of the receiver instance key and the parameter instance key. If the two instance keys come from the same method, we can first apply our local must-not-alias analysis (which is usually more precise). The analysis is given the associated local variables and assignment statements as input. If the local must-not-alias analysis determines that indeed those two variables at those statements cannot possibly point to the same object we return **true**, indicating that indeed the instance keys cannot alias.

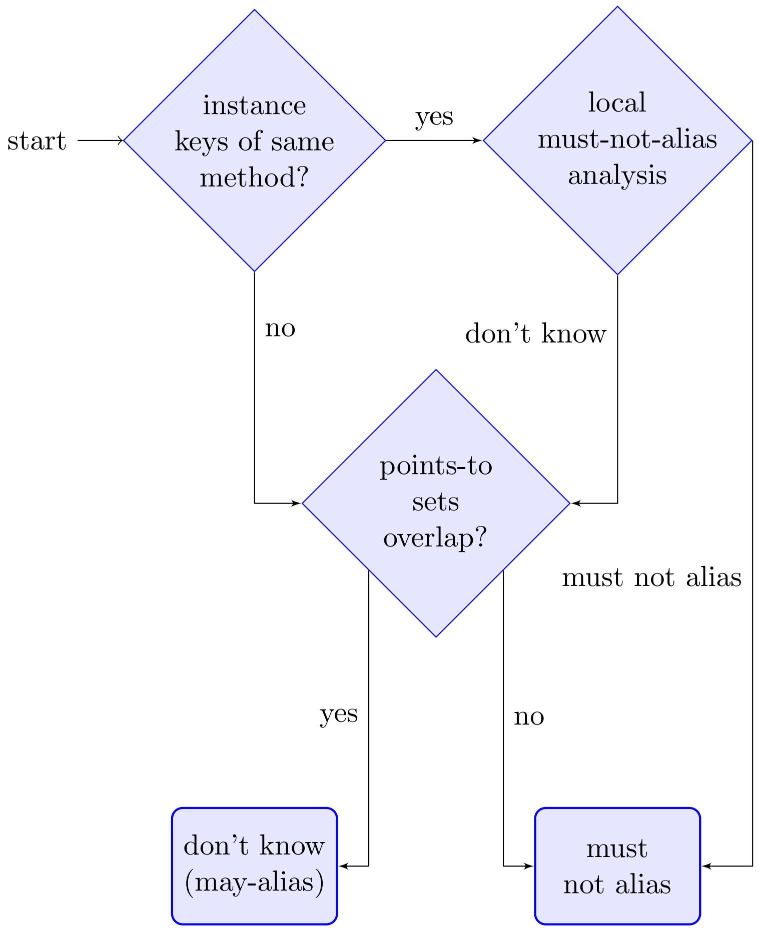

Figure 3: Decision procedure for the must-not-alias operation on instance keys

If the local 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. As Section 4 showed, sometimes the interprocedural analysis can be more precise, even for two variables coming from the same method. We construct points-to sets and refine them using context information for both instance keys. (In fact, we cache the refined points-to sets inside the instance keys). 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.

5.2.2 Implementing the method `mustAlias`

The implementation of the method `mustAlias(InstanceKey)` is similar and even more straightforward, because must-alias analysis cannot use our interprocedural points-to analysis. Hence, only local analysis is possible.

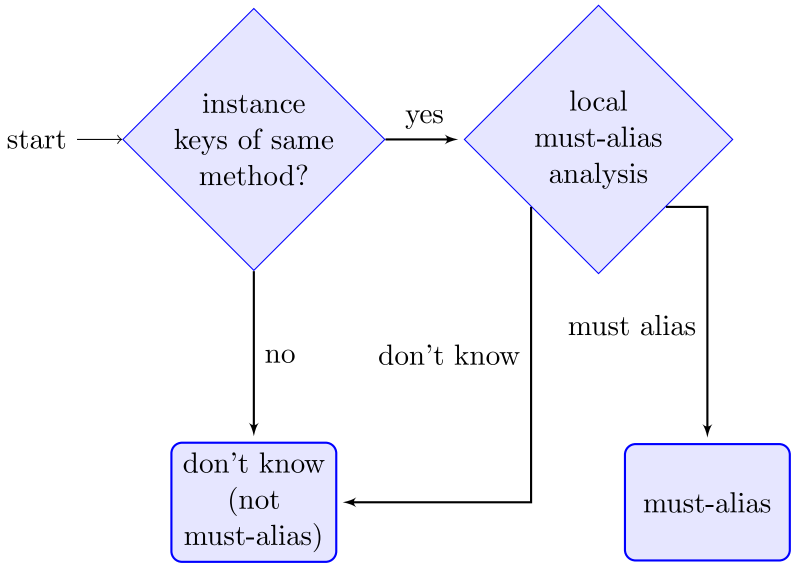
start
instance keys of same method?
yes
local must-alias analysis
no
don't know
must alias
don't know (not must-alias)
must-alias

Figure 4: Decision procedure for the must-alias operation on instance keys

Figure 4 presents a flowchart of our logic. First, we determine whether both instance keys represent values in the same method. If they don't, we simply cannot tell whether they must-alias and return `false`. Otherwise, we can delegate to the local must-alias analysis and return its answer. As we noted in Section 3.1, we implemented two different versions of the intraprocedural must-alias analysis with different semantics for redefinitions in loops; clients may select the must-alias analysis that best suits their needs.

Strong vs. weak instance keys The semantics of instance keys depends on which analyses are used. Strong instance keys implement their `mustAlias` method via a strong must-alias analysis, as defined in Section 3.1. Weak instance keys, on the other hand, use a weak must-alias analysis.

5.2.3 Implementation of `equals`

The implementation of the `equals(Object)` method is an important design decision. As mentioned above, it is very desirable to look up state from hash tables using instance keys. Since all hash maps and tables in the Java runtime library use the methods `hashCode()` and `equals(Object)` to look up associations, correctly implementing these methods is very important. Using instance keys, the semantics of `equals(Object)` fortunately becomes very simple. We already know that two instance keys, by definition, represent the same object if their associated variables must-alias. Hence, our implementation of the `equals` method simply checks that the object passed in is of type `InstanceKey`. If it is, it delegates to the method `mustAlias(InstanceKey)` and returns its result. Otherwise, it returns `false`.

By this mechanism it follows that the identity of an instance key is solely based on its must-alias relationship to other keys. In particular, strong and weak instance keys will have different semantics when being compared to other keys using `equals`. This is not accidental—it is intentional. As mentioned in Section 3.1, we have different client analysis for tracematches. Some of these analyses consider single loop iterations, while others consider multiple loop iterations. In general, client analyses can profitably use both weak and strong instance keys, depending on their particular needs.

Strong instance keys provide the strong guarantee that that two keys are only equal when they are known to represent the same object. This means that instance keys can transparently represent runtime objects, easing the design and implementation of abstract interpreters and other static analyses. On the other hand, strong instance keys need to be handled with care in the presence of loops. Consider the following example.

```
1 while(...) {
2     x = foo();
3      ...
4 }
```

In this code example, line 2 could create arbitrarily many objects and assign them to `x` over `x`'s lifetime. There is no compile-time upper bound to the number of objects assigned to `x`. For strong instance keys, we have to maintain the property that each key only represents one single concrete object. This would require that the instance key representing `x` at line 2 in this method cannot equal itself. (The respective strong must-alias analysis would assign $\top$ to `x`; remember that $\top$ also does not equal itself either).

It is common to define data flow analyses via a fixed point iteration. The fixed point is reached if the analysis state at any given statement after the n-th iteration is *equal* to the one in the previous iteration. But for strong instance keys representing values redefined in loops as above, `equals(..)` will always return `false`. Hence, a data flow analysis whose termination condition relies on instance keys being equal might never terminate. Data flow analyses using strong instance keys hence have to use an alternative termination condition. In our client analyses using strong instance keys, we capped the maximum number of iterations.

Note that this problem is not caused by the notion of instance keys. The problem is that the number of concrete objects assigned at runtime cannot be known at compile time. Hence, there can only be two choices: Approximate the unbounded number of objects by a finite number of instance keys—weak instance keys use a single key—or by an infinite number of strong instance keys. Developers may choose the most appropriate approach for their client analysis.

```
Method method;      //declaring method
String variable;    //associated variable name
Stmt stmt;          //associated statement
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result +
        method.fullName().hashCode();
    result = prime * result +
        must_alias_analysis(method).valNumber(variable,stmt);
    return result;
}
```

Figure 5: Implementation of the `hashCode()` method for instance keys (pseudo code)

5.2.4 Implementation of `hashCode`

For performance reasons, hash codes should differ whenever they can, but must be equal for two instance keys whenever an invocation of `equals` on those keys would return `true`, i.e. if both keys must be aliased. We found the following solution for computing an effective hash code for instance keys. The code is a function of

1. the identity of the declaring method; and
2. the value number assigned to the associated variable by the local must-alias analysis of this method at the associated statement.

Figure 5 gives our concrete implementation in Java-like pseudo code. This solution ensures that two instance keys from different methods are likely to be assigned different hash codes. Furthermore, instance keys representing potentially different values from the same method are likely to be assigned different keys. Moreover, keys representing the same value (with the same value number) at the same statement in the same method will be assigned the same hash code. Hence, this implementation is sound.

Note that this hash code can be cached, as its computation only depends on constant values. Our implementation in fact caches hash codes for improved efficiency. (In fact, in recent work researchers have convincingly argued [29] that any sound implementation of the `hashCode()` method should be a function of constant values.)

5.3 Experience

Our work on instance keys was inspired by our development of a static analysis for tracematches [3, 4]. Our initial static analysis did not use any flow-sensitive pointer information; it relied on whole-program pointer analysis as described in Section 4. The initial static analysis included an interprocedural stage that did not use any flow-sensitive information. This stage was not sufficiently precise to get any results at all. When we manually inspected our analysis results, we found that many cases would benefit from a flow-sensitive pointer analysis. In fact, we noticed that many of these cases only required flow-sensitive information at the intraprocedural level. At the interprocedural level, flow-insensitive information was sufficient for optimizing the cases that we were interested in.

Our initial implementations used variable names rather than instance keys for tracking heap objects, which led to the implementation-level that we explained. In particular, the same object can be pointed to by multiple different variables, which makes it difficult to look up analysis information. Furthermore, propagating variable names (in place of objects) can become unsound if the variables are redefined (potentially inside loops). Instance keys enabled us to clearly distinguish between program variables and the heap objects that they point to. The property that a strong instance key equals another exactly when it must alias the other one was very useful when it came to associating analysis states with instance keys, via hash maps.

6 Related work

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

6.1 Instance keys

We first came across instance keys in a series of papers by Fink et al. [9,26]. The authors 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 or how they are computed. We found that the notion of instance keys was useful in our own research, we explored this notion further and describe instance keys and their properties in this paper; we believe that instance keys will be helpful to others as well.

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, instance keys do support must-alias information, and strong instance keys can therefore be used in the place of runtime objects when implementing static analyses.

Generally, instance keys combine must and must-not aliasing information with interprocedural points-to information. Another way to combine must and must not information is to compute them together using a combined abstraction. Emami et al. [7] propose such a combined abstraction and integrate it with context-sensitive interprocedural pointer analysis, using the notation of abstract stack locations. Instance keys differ from abstract stack locations by giving names for objects on the heap, while abstract stack locations track stack-based objects.

6.2 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 [14], Hind surveys the most important research in this field, lists 11 open questions in points-to analysis, and points out 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 ones by Altucher and Landi [2] and by 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 instance keys 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. Instance keys were flexible enough to incorporate all these analyses.

6.3 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 [15] 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; many other client analyses are interesting. We next enumerate some interesting client analyses.

- Cast elimination [27] removes unnecessary casts for variables that are known to be assigned an object of the correct type at runtime.
- Side effect analysis [20] determines possible side effects of method calls (and side effects of advice for aspect-oriented programs). Many optimizations and abstract interpretations can profitably use the fact that a method is free of certain side effects, including parallelization.
- Escape analysis [23] determines if values may escape a given loop, method or class. Non-escaping values are particularly easy to handle and quite amenable to the intraprocedural techniques that we described in Section 3. Escape analyses are also useful for type-checking ownership types [6], which encode ownership constraints on aliases. For instance, some ownership type systems require that an object owned by some parent object is only ever be modified by that parent.
- Thread-local objects analysis [13] determines whether a given variable is used thread-locally, i.e. whether accesses to values stored in that variable will only ever be made by a single thread. Flow-sensitive client analyses often (implicitly) depend on the values not being modified by

concurrent threads, and require at least much more conservative assumptions in the presence of multiple threads.

- Static write barrier removal for generational garbage collectors [32] relies on intraprocedural must-alias information and a flow-insensitive interprocedural analysis to distinguish between heap objects. The design of this static analysis would appear to make it a good client for instance keys.

- Automatic parallelization always relies on pointer analysis, since different processors (or processor cores) work best if they do not have to simultaneously access the same part of the heap.

- A static analysis of PHP code for security vulnerabilities [16] needs to use pointer analysis to deal with aliasing between references. They have 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.

- Static race detection for Java requires pointer analysis, since Java locks and the state that they protect are both heap objects. Naik and Aiken therefore invented conditional must-not-alias analysis for this application [22].

- The LLVM compiler infrastructure [19] supports the development of client analyses. The infrastructure includes aggressive dead code elimination (which eliminates calls to pure functions whose results are thrown away); loop invariant code motion; and a transformation that converts calls-by-reference to calls-by-value, when it is safe. While the infrastructure supports value numbering, the interface for querying the alias information only takes two variables and determines whether they are aliased.

In general, all of the above client analyses work best with a maximally precise points-to or alias analysis. But these analyses 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 instance keys would help in the design and implementation of all of these analyses.

6.4 Specializing pointer analyses

In [25], Rountev et al. present an analysis that treats different parts of a program with different precision. Many programs usually use large standard libraries that are hard 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. Instance keys 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 [27] Sridharan and Bodík present a flow-insensitive, demand-driven, refinement-based 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 instance keys could support such an interface using parametrization: one instance key could be computed more carefully than usual, if it is particularly important to the client analysis.

7 Conclusions

In this paper we have described instance keys, a notation which integrates pointer analysis results from flow-insensitive interprocedural analyses with results from flow-sensitive intraprocedural analyses. Instance keys enable clients to determine whether two variables definitely point to the same heap object and whether two variables definitely point to different objects. Within the instance key framework we defined two different kinds of instance keys, *strong* and *weak*, that have different semantics with respect to the redefinition of variables within loops.

We believe that instance keys simplify the design and implementation of static analyses that rely on pointer analyses: with instance keys, 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 *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 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 Ernst [8], pages 525–549.

[4] E. Bodden, P. Lam, and L. Hendren. Flow-sensitive static optimizations for runtime monitoring. Technical Report abc-2007-3, http://www.aspectbench.org/, 07 2007.

[5] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, 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 *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 48–64, New York, NY, USA, 1998. ACM Press.

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

[8] 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*. Springer, 2007.

[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*, 2006.

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

[11] R. Ghiya and L. J. Hendren. Putting pointer analysis to work. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 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] R. L. Halpert, C. J. F. Pickett, and C. Verbrugge. Component-based lock allocation. In *PACT'07: Proceedings of the 16th International Conference on Parallel Architectures and Compilation Techniques*. IEEE, September 2007. To appear.

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

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

[16] N. Jovanovic, C. Kruegel, and E. Kirda. Precise alias analysis for syntactic detection of web application vulnerabilities. In *Proceedings of PLAS '06*, 2006.

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

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

[19] C. Lattner. The LLVM compiler system. In *Proceedings of the 2007 Bossa Conference on Open Source, Mobile Internet and Multimedia*, March 2007.

[20] A. Le, O. Lhoták, and L. Hendren. Using inter-procedural side-effect information in JIT optimizations. In R. Bodik, editor, *Compiler Construction, 14th International Conference*, 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, *Compiler Construction, 12th International Conference*, 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 ACM Symposium on Principles of Programming Languages (POPL '07)*, January 2007.

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

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

[25] A. Rountev, S. Kagan, and T. Marlowe. Interprocedural dataflow analysis in the presence of large libraries. In *International Conference on Compiler Construction*, LNCS 3923, pages 2–16, 2006.

[26] S. Shoham, E. Yahav, S. Fink, and M. Pistoia. Static specification mining using automata-based abstractions. In *ISSTA07*, pages 174–184, July 2007.

[27] M. Sridharan and R. Bodík. Refinement-based context-sensitive points-to analysis for Java. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 387–400, New York, NY, USA, 2006. ACM Press.

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

[29] M. Vaziri, F. Tip, S. Fink, and J. Dolby. Declarative object identity using relation types. In Ernst [8], pages 54–78.

[30] 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*, volume 1060 of *LNCS*, pages 74–90, April 1996.

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

[32] 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.