# MayEqual: A New Alias Question*

John Boyland†  Aaron Greenhouse‡

## Abstract

Any code-reordering transformation must take into account data dependencies. In object-oriented programs, data dependencies often exist through the use of fields of dynamically allocated objects. We propose that analyses supporting code reordering answer the following question: In the context of a particular block, could a pointer expression at one place in the block refer to the same object as a pointer expression in another place in the block. We show how a points-to analysis gives unnecessarily conservative answers to this question.

## 1 Introduction

Alias analysis can be defined as determining whether at a particular program point, two pointers refer to the same storage location. This information is used to determine data dependencies. One of the two standard aliasing questions can be phrased as follows (the other question is *MustAlias*):

> $MayAlias_{P[.]}(p_1, p_2)$: In some execution of $P$, do we have $p_1 = p_2$ at the indicated point $P[.]$?

Algorithmic analyses will return "true" if they cannot rule out equality. Interesting alias analyses for object-oriented languages can be obtained, for example, using Steensgard's "points-to analysis" (which is flow insensitive) or Sagiv, Reps and Wilhelm's "shape analysis" (which is flow sensitive).

In the ACT/Fluid project at CMU and UWM, we have been investigating when the effects of two computations may interfere, that is, when a possible data dependency may exist between two computations. In such cases, reordering may change the results of the program.[1] Part of the process involves asking whether a reference computed at one point could equal a reference computed at another point. For instance, if we want to reorder the first two statements in the following loop

```
while (...) { { ...; x.f = ...; ... }
              { ...; ... = y.f; ... }
              ... }
```

we must ensure that the expression `x` could never refer to the same object that is referred to by `y` (assuming both expressions evaluate).

At first glance, it may appear that *MayAlias* can be of assistance. However, at which program point(s) would the question be asked? The obvious choices are to ask the question at the program points where `x` and `y` are referenced in the above program fragment, but even if perfect answers to the *MayAlias* questions were available, they would not take into account whether the values of `x` and `y` *changed* between the two points. For example, consider the following scheme:

```
{ ...; x.f = ...; x = ...; }
{ y = ...; ... = y.f; ...; }
```

In general, the problem is that there is no point at which the question can be asked.[2]

The question to ask must also allow for useful flow sensitivity. Consider the following scheme:

```
{ if (x != y) { x.f = ...; } }
{ ...; ... = y.f; ...; }
```

*This material is based upon work supported by the Defense Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-2-0241.

†Address: Department of Electrical Engineering and Computer Science, University of Wisconsin-Milwaukee, Milwaukee, WI 53201. Email: `boyland@cs.uwm.edu`.

‡Address: Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213. Email: `aarong@cs.cmu.edu`.

[1] Our effort is complicated by the fact that the analysis must handle incomplete programs, using annotations on component interfaces as surrogates for actual code.

[2] Even if the code is put into SSA form, loops may prevent the value of `x` (say) from being available at the start of the second computation.

In this case, the condition around the first expression ensures there cannot be a data dependency (even though the second access is not similarly guarded).

Our taking issue with the *question* does not imply that any particular *analysis* is useless for determining whether the expressions could yield the same pointer value ("points-to" analysis, in particular, can provide useful, albeit conservative, answers). We wish only to present the fact that a new question is being asked—one that we believe allows the use of analyses that yield less conservative results.

## 2 A New Question

The question we wish analyses to answer is

$$\mathit{MayEqual}_{P[C[\cdot,\cdot]]}(e, e')$$

where $P$ is a program context with a hole for $C$, which is an execution context with two holes for expressions $e$ and $e'$. The question has the following meaning:

> Within some execution of $C$ in the program $P$, do the set of all values to which $e$ evaluates and the set of all values to which $e'$ evaluates have a nonempty intersection?

The inner context, $C$, is used to focus the analysis on the part of the program that includes the computations for which we are checking interference. When checking whether two computations interfere, $C$ should be chosen to be the smallest context that includes both computations. The importance of $C$ may be understood through the following intuition. Consider how the question might be answered for a *specific* execution of $C$. At the beginning of the block of code that is represented by $C$, two sets $s$ and $s'$ are initialized to be empty. Just before the expression represented by $e$, the object $e$ references is added to $s$, and just before the expression represented by $e'$, the object $e'$ references is added $s'$. At the end of the block of code represented by $C$, if $s$ and $s'$ have a non-empty intersection then $\mathit{MayEqual}_{P[C[\cdot,\cdot]]}(e, e')$ must be true.

Let us return to the `while` loop presented above to investigate how choices of $C$ affect the operation on whose behalf *MayEqual* is being asked; in this case the operation is statement reordering. As before, suppose we want to swap the order of the first two compound statements in the loop body. The best choices for the parameters to *MayEqual* are:

$$\begin{aligned} P &= \ldots\texttt{while (...)}\ \ \{\ \bullet\ \ldots\}\ldots \\ C &= \{\ldots;\bullet\texttt{.f} = \ldots;\ldots\}\{\ldots;\ldots = \bullet\texttt{.f};\ldots\} \\ e &= \texttt{x} \\ e' &= \texttt{y} \end{aligned}$$

With these contexts, if it is not the case that `y` may refer to same object as `x` at the two program points indicated in $C$, then the order of the two blocks may be swapped.[3] If instead we used the contexts

$$\begin{aligned} P' &= \ldots\bullet\ldots \\ C' &= \texttt{while (...) \{ \{ ...; }\bullet\texttt{.f = ...; ... \}} \\ &\phantom{=}\ \ \texttt{\{ ...; ... = }\bullet\texttt{.f; ... \}} \\ &\phantom{=}\ \ \texttt{... \}} \end{aligned}$$

then loop-carried dependencies could be mistakenly viewed as preventing the swapping of statements. Choosing a larger-than-necessary context yields conservative results. However, if we were interested in moving the second compound statement out of the loop, then our choice for $C$ must include the entire `while` statement.

Any analysis that answers the "points-to" question can be used to obtain a conservative answer to the *MayEqual* question with $P = \bullet$ and $C$ being the entire program with holes for the two expressions of interest. An example of a common situation for which a larger-than-necessary $C$ (such as provided by a "points-to" analysis) prevents reordering is if we wanted to ask questions about `x` and `y` in

```
while (...) {
  x = new Node(...);
  x.next = null;              -- (1)
  y.next = x;                 -- (2)
  y = x;
}
```

No matter how accurate a "points-to" analysis is, it must determine that `x` and `y` could point to the same location, and thus statements labeled (1) and (2) cannot be reordered.

[3] Here we assume there are no other potential data dependencies.