# Efficient Alias Set Analysis Using SSA Form

Nomair A. Naeem
D. R. Cheriton School of Computer Science
University of Waterloo, Canada
nanaeem@uwaterloo.ca

Ondřej Lhoták
D. R. Cheriton School of Computer Science
University of Waterloo, Canada
olhotak@uwaterloo.ca

## Abstract

Precise, flow-sensitive analyses of pointer relationships often represent each object using the set of local variables that point to it (the *alias set*), possibly augmented with additional predicates. Many such analyses are difficult to scale due to the size of the abstraction and due to flow sensitivity. The focus of this paper is on efficient representation and manipulation of the alias set. Taking advantage of certain properties of static single assignment (SSA) form, we propose an efficient data structure that allows much of the representations of sets at different points in the program to be shared. The transfer function for each statement, instead of creating an updated set, makes only local changes to the existing data structure representing the set. The key enabling properties of SSA form are that every point at which a variable is live is dominated by its definition, and that the definitions of any set of simultaneously live variables are totally ordered according to the dominance relation. We represent the variables pointing to an object using a list ordered consistently with the dominance relation. Thus, when a variable is newly defined to point to the object, it need only be added to the head of the list. A back edge at which some variables cease to be live requires only dropping variables from the head of the list. We prove that the analysis using the proposed data structure computes the same result as a set-based analysis. We empirically show that the proposed data structure is more efficient in both time and memory requirements than set implementations using hash tables and balanced trees.

***Categories and Subject Descriptors*** D.3.3 [*Programming Languages*]: Language Constructs and Features—Dynamic Storage Management; D.3.4 [*Programming Languages*]: Processors—Compilers; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program Analysis

***General Terms*** Algorithms, Languages, Performance, Verification

***Keywords*** dataflow analysis, pointer analysis, alias analysis, shape analysis, static single assignment form, dominance, live variables

## 1. Introduction

Many static analyses have been proposed to infer properties about the pointers created and manipulated in a program. The properties inferred by these analyses are useful in applications such as call graph construction, escape analysis, bug finding, and proving domain-specific correctness properties of the program. Different applications require different tradeoffs between precision and efficiency. This design space has been mapped and surveyed [12, 22]. A somewhat fuzzy distinction has been made between "shape" analyses (which are generally more precise) and "pointer" analyses (which are generally more efficient). A shape analysis emphasizes individual concrete objects and the relationships between them, whereas a pointer analysis emphasizes the pointers, and often models multiple concrete objects using the same abstract representative (e.g. an allocation site).

This paper focuses on an increasingly used abstraction, which we call alias sets, that combines certain aspects of both "pointer" and "shape" abstractions. An alias set is the set of all local pointer variables that point to a given object. The alias set contains all of the pointers that point to the object at run time, and no others. If the analysis is uncertain about whether a given pointer $x$ points to the concrete object, instead of creating a may- or must-point-to set, it creates two alias sets, only one of which contains $x$. As a result, like in a shape abstraction, every alias set (except the empty one) corresponds to at most one concrete object at any given point in time during program execution. This makes the abstraction precise and enables strong updates. Thus alias sets are useful for analyses that track individual objects. Like a pointer abstraction, an alias set emphasizes the local pointers pointing to the object, rather than the precise relationships between objects, which are expensive to model. The alias set abstraction is more precise than most pointer analyses in that it subsumes both may- and must-alias information. We will give a more precise definition and detailed discussion of alias sets in Section 2.

Alias sets have been found to be useful in several classes of applications. They are the basis of many shape abstractions. Sagiv et al. have designed a shape analysis that uses alias sets as the abstraction of a heap object, which it augments with edges between alias sets to model inter-object relationships [23]. Other shape analyses [11, 24] refine the object abstraction further (e.g. with heap access paths or domain-specific predicates), but generally retain the alias set at its core. Alias sets are a special case of access path sets: they are sets of access paths with zero dereferences. Whereas an access path begins at a local variable and specifies a list of fields to be followed to reach the object, an alias set contains only the local variables pointing directly to the object. Alias sets have also been successful in analyses that are lighter than shape analysis, but need more precise consideration of individual objects than most pointer analyses can provide. For example, the alias set abstraction has been used to detect memory leaks and automatically free objects [3, 19]. Alias sets have also been extended for precise checking of typestate properties [8, 9, 17]. For these applications, pointer analyses that determine only that a pointer points to some object allocated at a given allocation site are insufficient, because these



*ISMM'09,* June 19–20, 2009, Dublin, Ireland.

analyses need to keep track of individual objects as execution flows from one instruction to the next. We discuss these applications of alias sets in more detail in the related work section.

Alias set analysis subsumes both may- and must-alias analysis. Pointer analyses generally use one of two abstractions. The first are points-to pairs $(p, o)$, indicating that the pointer $p$ may point to one of the concrete objects represented by the abstract object $o$. The second are may- or must-alias pairs $(p_1, p_2)$, indicating that the pointers $p_1$ and $p_2$ may or must point to the same object. In comparison, the alias set abstraction associates with each program point a set of alias sets, each of the form $\{p_1, \ldots, p_n\}$. The presence of the set $\{p_1, \ldots, p_n\}$ indicates that there may exist an object pointed to by all of the pointers $p_1, \ldots, p_n$ and no others. The presence of an alias set containing both $p_1$ and $p_2$ at a given program point implies that $p_1$ and $p_2$ may be aliased at that point. On the other hand, if every alias set at a given program point contains either both $p_1$ and $p_2$ or neither of them, then $p_1$ and $p_2$ must be aliased at that point. If information about allocation sites is needed, an alias set could be augmented with an allocation site, and thus represent only those objects pointed to by the pointers in the set and allocated at the given allocation site.

In recent years Static Single Assignment (SSA) form [4] has gained popularity as an intermediate representation (IR) in optimizing compilers. The key feature of this IR is that every variable in the program is a target of only one assignment statement. Therefore, by construction, any use of a variable always has one reaching definition. This simplifies program analysis. SSA form has been applied in many compiler optimizations including value numbering, constant propagation and partial-redundancy elimination. In addition, SSA form has other less obvious properties that simplify program analysis. Specifically, the entire live range of any variable is dominated by the (unique) definition of that variable, and the definitions of any set of simultaneously live variables are totally ordered according to the dominance relation. Thus, the definition of one of the variables is dominated by all the others, and at this definition, the variables are all live and have the values that they will have until the end of the live range. These properties have been used to define an efficient register allocation algorithm [10]. We exploit these same properties to efficiently represent the set of variables pointing to an object.

Analyses using alias sets are difficult to make efficient for two reasons. First, the size of the abstraction is potentially exponential in the number of local variables that are ever simultaneously live. Second, the analyses are flow-sensitive, so many different alias sets must be maintained for different program points. The first issue, in the rare cases that the number of sets grows uncontrollably, can be effectively solved by one of several widenings suggested by Sagiv et al. [23]. Our work addresses the second issue. When the variable sets are represented using linked lists ordered by dominance, we show that due to the dominance properties of SSA form, updates needed to implement the analysis occur only at the head of the lists. As a result, tails of the lists can be shared for different program points.

This paper makes the following contributions:

- We formalize an alias set abstraction for programs in SSA form. The abstraction can be implemented using any set data structure, including ordered lists. The abstraction can be used to provide may and must-alias information to a client analysis or used in a shape analysis with or without further information about incoming pointers from other objects.
- We prove that if the program being analyzed is in SSA form and if the lists are ordered according to the dominance relation on the definition sites of variables, then the analysis requires only local updates at the head of each list. Thus, the tails of the lists can be shared at different program points.
- We implement an interprocedural context-sensitive analysis using the abstraction as an instance of the IFDS algorithm [20], and evaluate the benefits of the list-based data structure compared to sets implemented using balanced trees and hash tables.

The remainder of the paper is organized as follows: Section 2 formalizes the alias set abstraction and defines transfer functions that can be used in any standard dataflow analysis algorithm to compute the abstraction. In Section 3 we give a brief introduction to SSA form and define terms used in the remainder of the paper. Section 4 presents a new data structure and corresponding transfer functions for representing alias sets. Empirical results comparing the running times and memory consumption of the analysis using different data structures are presented in Section 5. We discuss related work in Section 6 and give concluding remarks in Section 7.

## 2. Alias Set Analysis

This section defines how objects are represented using alias sets and presents a transfer function to determine the alias sets at each program point.

The overall abstraction $\rho^\sharp$ is a set of abstract objects (i.e. alias sets). This abstract set is an overapproximation of run-time behaviour. For every concrete object that could exist at run time at a given program point, the abstraction always contains an alias set that abstracts that concrete object; however, the abstraction may conservatively contain additional alias sets that do not correspond to any concrete object. Each alias set $o^\sharp$ is a set of local variables of pointer type. The alias set contains exactly those variables that point to the corresponding concrete object at run time. The alias set is neither a may-point-to nor a must-point-to approximation of the concrete object; it contains all pointers that point to the concrete object and no others. If the analysis is uncertain whether a given pointer $x$ points to the concrete object, it must represent the concrete object with two alias sets, one containing $x$ and the other not containing $x$.

For example, consider a concrete environment in which variables $x$ and $y$ point to distinct objects and $z$ may be either null or point to the same object as $x$. The abstraction of this environment would be the set of alias sets $\{\{x\}, \{x, z\}, \{y\}\}$.

Each alias set except the empty set represents at most one concrete object at any given instant at run time. For example, consider the alias set $\{x\}$. At run time, the pointer $x$ can only point to one concrete object $o$ at a time; thus at that instant, the alias set $\{x\}$ represents only $o$ and no other concrete objects. This property enables very precise transfer functions for individual alias sets, with strong updates. Continuing the example, the program statement `y := x` transforms the alias set $\{x\}$ to $\{x, y\}$, with no uncertainty. We know that the unique concrete object represented by $\{x\}$ before the statement is represented by $\{x, y\}$ after the statement. Of course, since the analysis is conservative, there may be other spurious alias sets in the abstraction. The important point is that any given abstract object is tracked precisely by the analysis.

This basic abstraction can be extended or refined as appropriate for specific analyses. For example, Sagiv et al. [23] define a shape analysis that uses this same abstraction to represent objects, and adds edges between abstract objects to represent pointer relationships between concrete objects. Other analyses refine the abstraction by adding conditions to the alias sets that further limit the concrete objects that they represent. For example, an alias set representing concrete objects pointed to by a given set of pointers can be refined to represent only those concrete objects that were also allocated at a given allocation site.

$$
\begin{aligned}
[\![s]\!]^1_{\text{gen}} &\triangleq \begin{cases} \{\{v\}\} & \text{if } s = v \leftarrow \textbf{new} \\ \emptyset & \text{otherwise} \end{cases} \\
[\![s]\!]^1_{o^\sharp}(o^\sharp) &\triangleq \begin{cases} \{o^\sharp \cup \{v_1\}\} & \text{if } s = v_1 \leftarrow v_2 \wedge v_2 \in o^\sharp \\ \{o^\sharp \setminus \{v_1\}\} & \text{if } s = v_1 \leftarrow v_2 \wedge v_2 \notin o^\sharp \\ \{o^\sharp \setminus \{v\}\} & \text{if } s \in \{v \leftarrow \textbf{null}, v \leftarrow \textbf{new}\} \\ \{o^\sharp\} & \text{if } s = e \leftarrow v \\ \{o^\sharp \setminus \{v\}, o^\sharp \cup \{v\}\} & \text{if } s = v \leftarrow e \end{cases} \\
[\![s]\!]^1_{\rho^\sharp}(\rho^\sharp) &\triangleq [\![s]\!]^1_{\text{gen}} \cup \bigcup_{o^\sharp \in \rho^\sharp} [\![s]\!]^1_{o^\sharp}(o^\sharp)
\end{aligned}
$$

**Figure 1.** Transfer functions on individual alias sets. The superscript $^1$ on the function identifies the version of the transfer function; we will present modified versions of the transfer functions later in the paper.

The abstraction subsumes both may-alias and must-alias relationships. If variables $x$ and $y$ point to distinct objects, $\rho^\sharp$ will not contain any set containing both $x$ and $y$. If variables $x$ and $y$ point to the same object, every set in $\rho^\sharp$ will contain either both $x$ and $y$, or neither of them.

The analysis is performed on a simplified intermediate representation containing the following intraprocedural instructions:

$$s ::= v_1 \leftarrow v_2 \mid v \leftarrow \textbf{e} \mid \textbf{e} \leftarrow v \mid v \leftarrow \textbf{null} \mid v \leftarrow \textbf{new}$$

The symbol **e** represents any heap location, such as a field of an object or an array element and $v$ can be any variable from the set of local variables of the current method. The instructions are self-explanatory: they copy object references between variables and the heap, assign the **null** reference to a variable, and create a new object. In addition, the IR contains method call and return instructions.

In Figure 1 we define transfer functions that specify the effect of an instruction on a single alias set at a time. If $s$ is any statement in the IR except a heap load, and if $o^\sharp$ is the set of variables pointing to a given concrete object $o$, then it is possible to compute the exact set of variables which will point to $o$ after the execution of $s$. This enables the analysis to flow-sensitively track individual objects along control flow paths. When a new object is created and assigned to a variable ($s = v \leftarrow$ **new**), the transfer function $[\![s]\!]^1_{\text{gen}}$ creates a new alias set containing only $v$, since at runtime, after $s$ executes, $v$ is the only variable that points to the new object. The copy statement $v_1 \leftarrow v_2$ either adds or removes the variable $v_1$ from an alias set depending on whether the source variable $v_2$ points to the object. Strong updates are also performed in the case of **null** and **new** since, after these assignments execute, the assigned variable no longer points to any object that it was previously pointing to. The store statement has no effect on an alias set, since it does not affect the values of any local variables. A load splits an alias set into two, one containing the target of the load and the other not containing it. The overall transfer function $[\![s]\!]^1_{\rho^\sharp}$ applies the per-alias-set transfer function $[\![s]\!]^1_{o^\sharp}$ to each alias sets in the abstract environment.

The alias sets at each point in the program can be computed using these transfer functions in a standard worklist-based dataflow analysis framework like the one shown in Algorithm 1. The analysis is a forward dataflow analysis where the elements of the lattice are the abstract environments, $\rho^\sharp$ (i.e., sets of alias sets). The merge operation is set union.

**Algorithm 1**: Dataflow Analysis

```
for each statement s, initialize out[s] to ⊥
add all statements to worklist
while worklist not empty do
    remove some s from worklist
    in = ⊔_{p∈pred(s)} out[p]
    out[s] = [[s]]_{ρ♯}( in )
    if out[s] has changed then
        foreach s' ∈ succs(s) do
            add s' to worklist
    end
end
```

## 3. Static Single Assignment (SSA) Form

The key feature of Static Single Assignment (SSA) form [4] is that every variable in the program is a target of only one assignment statement. Therefore, by construction, any use of a variable always has one reaching definition.

Converting a program into SSA form requires a new kind of instruction to be added to the intermediate representation. At each control flow merge point with different reaching definitions of a variable on the incoming edges, a $\phi$ instruction is introduced to select the reaching definition corresponding to the control flow edge taken to reach the merge. The selected value is assigned to a freshly-created variable, thereby preserving the single assignment property. If multiple variables require $\phi$ nodes at a given merge point, the $\phi$ nodes for all the variables are to be executed simultaneously. To emphasize this point, we will group all $\phi$ nodes at a given merge point into one multi-variable $\phi$ node:

$$\begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix} = \phi \begin{pmatrix} x_{11} & \cdots & x_{1n} \\ \vdots & & \vdots \\ x_{m1} & \cdots & x_{mn} \end{pmatrix}$$

Each row, $i$, on the right side represents $n$ reaching definitions of variable $x_i$. When control reaches the $\phi$ instruction through some predecessor $p$ (with $1 \leq p \leq n$) of the $\phi$ instruction then the $p^{th}$ column of the right side defines the values to be assigned to the $y_i$ variables on the left side in a simultaneous parallel assignment. Given a *phi* function $\phi$ and a predecessor $p$, we write $\sigma(\phi, p)$ to denote this parallel assignment:

$$\sigma(\phi, p) = \begin{pmatrix} y_1 \leftarrow x_{1p} \\ \vdots \\ y_m \leftarrow x_{mp} \end{pmatrix}$$

We now present some standard definitions. An instruction $a$ *dominates* instruction $b$ if every path from the entry point to $b$ passes through $a$. We denote the set of instructions that domi-

nate instruction $s$ by dom($s$). By definition every instruction dominates itself. We write sdom($s$) to denote the set of instructions that *strictly* dominate $s$ i.e. dom$(s) \setminus \{s\}$. The *immediate* dominator of an instruction $s$, idom($s$), is an instruction in sdom($s$) dominated by every instruction in sdom($s$). It is well known that every instruction except the entry point has a unique immediate dominator. We use the notation defs($s$) to denote the set of variables defined (i.e written to) by the instruction $s$ and vars($S$) to denote the set of variables defined by the instructions in a set $S$ (i.e. vars$(S) \triangleq \bigcup_{s \in S}$ defs$(s)$).

In adapting the alias set analysis to work on SSA form, we do not want to reduce its precision. It is well known that as long as the transfer function is distributive, the fixed point computed by Algorithm 1 is equal to the merge-over-all-paths (MOP) dataflow value [14]. The transfer function $[\![s]\!]^1_{\rho^\sharp}$ for alias sets is indeed distributive:

$$\begin{aligned} & [\![s]\!]^1_{\rho^\sharp}(\rho^\sharp_1) \sqcup [\![s]\!]^1_{\rho^\sharp}(\rho^\sharp_2) \\ = \; & [\![s]\!]^1_{\text{gen}} \cup \bigcup_{o^\sharp \in \rho^\sharp_1} [\![s]\!]^1_{o^\sharp}(o^\sharp) \cup \bigcup_{o^\sharp \in \rho^\sharp_2} [\![s]\!]^1_{o^\sharp}(o^\sharp) \\ = \; & [\![s]\!]^1_{\text{gen}} \cup \bigcup_{o^\sharp \in \rho^\sharp_1 \cup \rho^\sharp_2} [\![s]\!]^1_{o^\sharp}(o^\sharp) \\ = \; & [\![s]\!]^1_{\rho^\sharp}(\rho^\sharp_1 \sqcup \rho^\sharp_2) \end{aligned}$$

Thus we would like to compute the MOP value even when analyzing the code in SSA form. However, blindly applying Algorithm 1 to SSA form will give a less precise value because of a unique semantic property of $\phi$ instructions: the effect of a $\phi$ instruction depends on which incoming control flow edge is used to reach it. Algorithm 1 ignores this property, in that it merges the input values from all incoming control flow edges before applying the transfer function. Therefore, when applied to a $\phi$ instruction, the algorithm applies *all* of the parallel assignments associated with *all* of the incoming control flow edges to the incoming dataflow value on *each* edge. That is, it conservatively applies parallel assignments to dataflow values on edges to which they should not have been applied, reducing precision. To eliminate this imprecision, we use a modified version of the algorithm (Algorithm 2) that handles $\phi$ instructions as a special case, respecting their unique semantics. The transfer function for a $\phi$ instruction, rather than depending only on the incoming dataflow value, is modified to also take the control flow predecessor as a parameter. As a result, the transfer function applies the parallel assignment associated with only that specific predecessor, preserving the precision of the dataflow analysis on the original intermediate representation. Therefore, since the transfer functions are distributive, when Algorithm 2 is applied to code in SSA form, it computes the MOP alias sets for the original code.

We emphasize that none of the theoretical properties discussed in the remainder of this paper rely on the use of Algorithm 2 to perform the dataflow analysis. All of the optimizations that we perform on the transfer functions would still be valid if we used Algorithm 1 to perform the dataflow analysis. However, because Algorithm 1 analyzes $\phi$ instructions in an imprecise way, the result would not necessarily be as precise as the MOP result on the original form of the code.

## 4. Efficient Alias Set Representation

In this section, we first extend the alias set analysis to work on SSA form. We then take advantage of the properties of SSA form to propose an efficient representation of alias sets using shared linked lists that require only local updates at the head of the list.

To extend the transfer function from Figure 1 to SSA form, we define it for $\phi$ instructions as shown in Figure 2. As discussed in the previous section, the transfer function for a $\phi$ instruction has two parameters: the incoming dataflow value $\rho^\sharp$ and the control flow predecessor $p$ in which the dataflow value arrives. The transfer function first determines the parallel assignment $\sigma(\phi, p)$ that corresponds to the given incoming control flow edge $p$. The alias set is then updated by adding all destination variables whose values are being assigned from variables already in the alias set, and removing all variables whose values are being assigned from variables *not* in the alias set. Notice that the transfer function for the simple assignment statement $v_1 \leftarrow v_2$ is a special case of the transfer function for $\phi$ when the parallel assignment $\sigma$ contains only the single assignment $v_1 \leftarrow v_2$.

**Algorithm 2**: Dataflow Analysis for SSA Form

```
for each statement s, initialize out[s] to ⊥
add all statements to worklist
while worklist not empty do
    remove some s from worklist
    if s is a φ instruction then
        foreach p ∈ preds(s) do
            out[s] = out[s] ⊔ [[φ]]_ρ♯(out[p], p)
    else
        in = ⨆_{p∈pred(s)} out[p]
        out[s] = [[s]]_ρ♯( in )
    end
    if out[s] has changed then
        foreach s' ∈ succs(s) do
            add s' to worklist
    end
end
```

$$[\![\phi]\!]^1_{o^\sharp}(o^\sharp, p) \triangleq \left\{ \begin{array}{r} o^\sharp \cup \{y_i : y_i \leftarrow x_i \in \sigma(\phi, p) \wedge x_i \in o^\sharp\} \\ \setminus \{y_i : y_i \leftarrow x_i \in \sigma(\phi, p) \wedge x_i \notin o^\sharp\} \end{array} \right\}$$

$$[\![\phi]\!]^1_{\rho^\sharp}(\rho^\sharp, p) \triangleq \bigcup_{o^\sharp \in \rho^\sharp} [\![\phi]\!]^1_{o^\sharp}(o^\sharp, p)$$

**Figure 2.** Transfer function for the $\phi$ instruction

For convenience, we transform the IR by inserting a trivial $\phi$ instruction with zero variables at every merge point that does not already have a $\phi$ instruction. In the resulting control flow graph, all statements other than $\phi$ instructions have only one predecessor.

In the remainder of this section we make use of SSA properties to derive a new representation of alias sets in a program. In Section 4.1 we make use of the liveness property of programs in SSA form to simplify the transfer functions presented so far. Section 4.2 presents a data structure which makes it possible to implement the simplified transfer functions efficiently. Finally in Section 4.3 we discuss further techniques to make the data structure efficient in both time and memory.

### 4.1 Live variables

In the alias set abstraction presented so far, the representation of an object was the set of all local variables pointing to it. However, applications of the analysis only ever need to know which *live* variables are pointing to the object. If a variable is not live, then its current value will never be read, so its current value is irrelevant. Thus, it is safe to remove any non-live variables from the alias set. This reduces the size of each alias set $o^\sharp$, and may even reduce the number of such sets in $\rho^\sharp$, since sets that differ only in non-live variables can be merged. One way to achieve this improvement is

to perform a liveness analysis before the alias set analysis, then intersect each alias set computed by the transfer function with the set of live variables, as shown in the revised transfer function in Figure 3.

$$\begin{aligned}
\text{filter}(\ell, \rho^\sharp) &\triangleq \{o^\sharp \cap \ell : o^\sharp \in \rho^\sharp\} \\
[\![s]\!]^2_{\rho^\sharp}(\rho^\sharp) &\triangleq \text{filter}(\text{live-out}(s), [\![s]\!]^1_{\rho^\sharp}(\rho^\sharp)) \\
[\![\phi]\!]^2_{\rho^\sharp}(\rho^\sharp, p) &\triangleq \text{filter}(\text{live-out}(\phi), [\![\phi]\!]^1_{\rho^\sharp}(\rho^\sharp, p))
\end{aligned}$$

**Figure 3.** Transfer function with liveness filtering

The irrelevance of non-live variables enables us to take advantage of the following property of SSA form:

**Property 1.** *If variable $v$ is live-out at instruction $s$, then the definition of $v$ dominates $s$.*

This property implies that the set of live variables is a subset of the variables whose definitions dominate the current program point. That is, for every instruction $s$, live-out$(s) \subseteq$ vars(dom$(s)$). Thus it is safe to intersect the result of each transfer function with the larger set vars(dom$(s)$), as shown in the modified transfer function in Figure 4.

$$\begin{aligned}
[\![s]\!]^3_{\rho^\sharp}(\rho^\sharp) &\triangleq \text{filter}(\text{vars}(\text{dom}(s)), [\![s]\!]^1_{\rho^\sharp}(\rho^\sharp)) \\
[\![\phi]\!]^3_{\rho^\sharp}(\rho^\sharp, p) &\triangleq \text{filter}(\text{vars}(\text{dom}(\phi)), [\![\phi]\!]^1_{\rho^\sharp}(\rho^\sharp, p))
\end{aligned}$$

**Figure 4.** Transfer function with dominance filtering

In order to simplify the transfer functions further, we will need the following lemma, which states that the alias sets returned by the original transfer function from Figures 1 and 2 contain only variables defined in the statement being abstracted and variables contained in the incoming alias sets.

**Lemma 1.** *Define vars$(\rho^\sharp) = \bigcup_{o^\sharp \in \rho^\sharp} o^\sharp$. Then:*

- *vars$([\![s]\!]^1_{\rho^\sharp}(\rho^\sharp)) \subseteq$ vars$(\rho^\sharp) \cup$ defs$(s)$, and*
- *vars$([\![\phi]\!]^1_{\rho^\sharp}(\rho^\sharp, p)) \subseteq$ vars$(\rho^\sharp) \cup$ defs$(\phi)$.*

*Proof.* By case analysis of the definition of $[\![s]\!]^1$ and $[\![\phi]\!]^1$. □

Recall that the IR has been transformed so that every non-$\phi$ instruction $s$ has a unique predecessor $p$. Since $p$ is the only predecessor of $s$, dom$(p) =$ sdom$(s)$. Therefore, as long as the output dataflow set for $p$ is a subset of dom$(p)$, the input dataflow set for $s$ is a subset of sdom$(s)$. By Lemma 1, the output dataflow set for $s$ is therefore a subset of vars(sdom$(s)) \cup$ defs$(s) =$ vars(dom$(s)$). Thus, the filtering using vars(dom$(s)$) is redundant. That is, the transfer functions shown in Figure 5 have the same least fixed point solution as the transfer functions from Figure 4. This is formalized in Theorem 1.

$$\begin{aligned}
[\![s]\!]^4_{\rho^\sharp}(\rho^\sharp) &\triangleq [\![s]\!]^1_{\rho^\sharp}(\rho^\sharp) \\
[\![\phi]\!]^4_{\rho^\sharp}(\rho^\sharp, p) &\triangleq \text{filter}(\text{vars}(\text{dom}(\phi)), [\![\phi]\!]^1_{\rho^\sharp}(\rho^\sharp, p))
\end{aligned}$$

**Figure 5.** Simplified transfer function with dominance filtering

**Theorem 1.** *Algorithm 2 produces the same result when applied to the transfer functions in Figure 5 as when applied to the transfer functions in Figure 4.*

*Proof.* It suffices to prove that when the algorithm is applied to the transfer function in Figure 5, every set in out$[s]$ is a subset of vars(dom(s)). This is proved by induction on $k$, the number of iterations of the algorithm. Initially, the out sets are all empty, so the property holds in the base case $k = 0$. Assume the property holds at the beginning of an iteration. If the iteration processes a non-$\phi$ instruction, Lemma 1 ensures that the property is preserved at the end of the iteration. If the iteration processes a $\phi$ instruction, the definition of $[\![\phi]\!]^4_{\rho^\sharp}$ ensures that the property is preserved at the end of the iteration. □

**Corollary 1.** *When Algorithm 2 runs on the transfer functions from Figure 4 or Figure 5, the transfer function $[\![s]\!]_{\rho^\sharp}$ is evaluated only on alias sets that are subsets of vars(sdom$(s)$).*

Due to Corollary 1, the set difference operations in $[\![s]\!]^1_{o^\sharp}$ are now redundant. Thus, the simplified transfer function $[\![s]\!]^5_{o^\sharp}$ shown in Figure 6 computes the same result as $[\![s]\!]^4_{o^\sharp}$.

The transfer function for $\phi$ instructions can be simplified in a similar way. If we intersect $[\![\phi]\!]^1_{o^\sharp}(o^\sharp, p)$ with vars(dom$(\phi)$), the definition from Figure 2 can be rewritten as:

$$\begin{aligned}
[\![\phi]\!]^5_{o^\sharp}(o^\sharp, p) &\triangleq \left\{ \begin{array}{l} o^\sharp \setminus \{y_i : y_i \leftarrow x_i \in \sigma \wedge x_i \notin o^\sharp\} \\ \quad \cup \{y_i : y_i \leftarrow x_i \in \sigma \wedge x_i \in o^\sharp\} \\ \quad \cap \text{vars}(\text{dom}(\phi)) \end{array} \right\} \\
&= \left\{ \begin{array}{l} o^\sharp \setminus \text{defs}(\phi) \\ \quad \cup \{y_i : y_i \leftarrow x_i \in \sigma \wedge x_i \in o^\sharp\} \\ \quad \cap (\text{defs}(\phi) \cup \text{vars}(\text{sdom}(\phi))) \end{array} \right\} \\
&= \left\{ \begin{array}{l} o^\sharp \cap \text{vars}(\text{sdom}(\phi)) \\ \quad \cup \{y_i : y_i \leftarrow x_i \in \sigma \wedge x_i \in o^\sharp\} \end{array} \right\}
\end{aligned}$$

We summarize the results of this section as follows:

**Theorem 2.** *Algorithm 2 produces the same result when applied to the transfer functions in Figure 6 as when applied to the transfer functions in Figure 4.*

*Proof.* By Theorem 1 and the reasoning in the two preceding paragraphs. □

Corollary 1 also applies to the transfer functions in Figure 6.

### 4.2 Variable Ordering

In the preceding section, we simplified the transfer function so that it performs only two operations on sets of alias sets. The first operation is adding a variable defined in the current instruction to an alias set. The second operation is intersecting each alias set with vars(sdom$(\phi)$), where $\phi$ is the current instruction. In this section, we present a data structure that makes it possible to implement each of these operations efficiently. The data structure is an ordered linked list with a carefully selected ordering. To construct the ordering, we take advantage of the following property of the dominance tree.

**Property 2.** *Suppose the instructions in a procedure are numbered in ascending order in a preorder traversal of the dominance tree. Then whenever instruction $s_1$ dominates instruction $s_2$, the preorder number of $s_1$ is smaller than the preorder number of $s_2$.*

If the program is in SSA form, we can extend the numbering of instructions to a numbering of variables in the program by numbering each variable when its unique definition is visited in traversing the dominance tree. A single $\phi$ instruction may define multiple variables; in this case, we number these variables in an arbitrary but fixed order. Parameters of the procedure, which are

$$
\begin{aligned}
[\![s]\!]^5_{\text{gen}} &\triangleq \begin{cases} \{\{v\}\} & \text{if } s = v \leftarrow \textbf{new} \\ \emptyset & \text{otherwise} \end{cases} \\
[\![s]\!]^5_{o^\sharp}(o^\sharp) &\triangleq \begin{cases} \{o^\sharp \cup \{v_1\}\} & \text{if } s = v_1 \leftarrow v_2 \wedge v_2 \in o^\sharp \\ \{o^\sharp, o^\sharp \cup \{v\}\} & \text{if } s = v \leftarrow e \\ \{o^\sharp\} & \text{otherwise} \end{cases} \\
[\![s]\!]^5_{\rho^\sharp}(\rho^\sharp) &\triangleq [\![s]\!]^5_{\text{gen}} \cup \bigcup_{o^\sharp \in \rho^\sharp} [\![s]\!]^5_{o^\sharp}(o^\sharp) \\
[\![\phi]\!]^5_{o^\sharp}(o^\sharp, p) &\triangleq \left\{ \left( o^\sharp \cap \text{vars}(\text{sdom}(\phi)) \right) \cup \{ y_i : y_i \leftarrow x_i \in \sigma(\phi, p) \wedge x_i \in o^\sharp \} \right\} \\
[\![\phi]\!]^5_{\rho^\sharp}(\rho^\sharp, p) &\triangleq \bigcup_{o^\sharp \in \rho^\sharp} [\![\phi]\!]^5_{o^\sharp}(o^\sharp, p)
\end{aligned}
$$

**Figure 6.** Transfer functions without set difference operations

$$
\begin{aligned}
[\![s]\!]^6_{\text{gen}} &\triangleq \begin{cases} \{\textbf{cons}(v, \textbf{empty})\} & \text{if } s = v \leftarrow \textbf{new} \\ \textbf{empty} & \text{otherwise} \end{cases} \\
[\![s]\!]^6_{o^\sharp}(o^\sharp) &\triangleq \begin{cases} \{\textbf{cons}(v_1, o^\sharp)\} & \text{if } s = v_1 \leftarrow v_2 \wedge v_2 \in o^\sharp \\ \{o^\sharp, \textbf{cons}(v, o^\sharp)\} & \text{if } s = v \leftarrow e \\ \{o^\sharp\} & \text{otherwise} \end{cases} \\
[\![s]\!]^6_{\rho^\sharp}(\rho^\sharp) &\triangleq [\![s]\!]^6_{\text{gen}} \cup \bigcup_{o^\sharp \in \rho^\sharp} [\![s]\!]^6_{o^\sharp}(o^\sharp) \\
\text{prune}(o^\sharp, \phi) &= \begin{cases} \textbf{empty} & \text{if } o^\sharp = \textbf{empty} \\ o^\sharp & \text{if } \text{car}(o^\sharp) \in \text{vars}(\text{sdom}(\phi)) \\ \text{prune}(cdr(o^\sharp), \phi) & \text{otherwise} \end{cases} \\
[\![\phi]\!]^6_{o^\sharp}(o^\sharp, p) &\triangleq \{ \text{ foldl } (\text{cons}, \text{prune}(o^\sharp, \phi), \{ y_i : y_i \leftarrow x_i \in \sigma \wedge x_i \in o^\sharp \}) \} \\
[\![\phi]\!]^6_{\rho^\sharp}(\rho^\sharp, p) &\triangleq \bigcup_{o^\sharp \in \rho^\sharp} [\![\phi]\!]^6_{o^\sharp}(o^\sharp, p)
\end{aligned}
$$

**Figure 7.** Transfer functions on sorted lists

all defined in the start node, are numbered in the same way. The resulting numbering has the property that if the definition of $v_1$ dominates the definition of $v_2$, then $\text{prenum}(v_1) < \text{prenum}(v_2)$.

To represent each alias set, we use a linked list of variables sorted in decreasing prenumber order. We will show that the two operations needed to implement the transfer function manipulate only the head of the list.

Recall from Corollary 1 that the transfer function for non-$\phi$ statements is only applied to alias sets that are a subset of $\text{vars}(\text{sdom}(s))$, where $s$ is the statement for which the transfer function is being computed. To process a $\phi$ statement, the transfer function shown in Figure 6 first intersects each incoming alias set with $\text{vars}(\text{sdom}(\phi))$, then adds variables defined in $\phi$ to it. In both cases, variables defined in the current statement $s$ are being added to a set that is a subset of $\text{vars}(\text{sdom}(s))$. Thus the definition of each variable being added is dominated by the definition of every variable in the existing set. Therefore, adding the new variables to the head of the list representing the set preserves the decreasing prenumber ordering of the list.

Now consider the intersection $o^\sharp \cap \text{vars}(\text{sdom}(\phi))$ that occurs in the transfer function for a $\phi$ instruction. The incoming alias set $o^\sharp$ is in the out set of one of the predecessors $p$ of $\phi$. Therefore, due to Theorem 2, $o^\sharp \subseteq \text{vars}(\text{dom}(p))$. We use the following property of dominance to relate $\text{vars}(\text{dom}(p))$ to $\text{vars}(\text{sdom}(\phi))$.

**Property 3.** *Suppose instructions $a$ and $b$ both dominate instruction $c$. Then either $a$ dominates $b$ or $b$ dominates $a$.*

Since any path to $p$ can be extended to be a path to $\phi$, every strict dominator of $\phi$ dominates $p$. Thus, $\text{sdom}(\phi) \subseteq \text{dom}(p)$. Let $a$ be any instruction in $\text{dom}(p) \backslash \text{sdom}(\phi)$. The instruction $a$ cannot dominate any instruction $b \in \text{sdom}(\phi)$, since by transitivity of dominance, it would then dominate $\phi$. By Property 3, every instruction in $\text{sdom}(\phi)$ dominates $a$. Therefore, $a$ has a higher preorder number than any instruction in $\text{sdom}(\phi)$, so $a$ appears earlier in the list representing $o^\sharp$ than any instruction in $\text{vars}(\text{sdom}(\phi))$. Therefore, to compute $o^\sharp \cap \text{vars}(\text{sdom}(\phi))$, we need only drop elements from the head of the list until the head of the list is in $\text{vars}(\text{sdom}(\phi))$. This is done using the prune function in Figure 7. The rest of Figure 7 gives an implementation of the transfer functions from Figure 6 using ordered lists to represent alias sets. Adding a variable to a set has been replaced by cons, which adds the variable to the head of the list, and intersection with $\text{vars}(\text{sdom}(\phi))$ has been replaced by a call to prune.

### 4.3 Data Structure Implementation

To further reduce the memory requirements of the analysis, we use hash consing to maximize sharing of cons cells between lists. Hash consing ensures that two lists with the same tail share that tail. In our implementation, we define an `HCList`, which can either be the empty list or a `ConsCell`, which contains a variable and a tail of

type HCList. We maintain a map **Var** × HCList → ConsCell. Whenever the analysis performs a cons operation, the map is first checked for an existing cell with the same variable and tail. If such a cell exists, it is reused instead of a new one being created.

## 5. Empirical Evaluation

We have extended the analysis defined in the preceding sections to a context-sensitive interprocedural analysis by implementing it as an instance of the interprocedural finite distributive subset (IFDS) algorithm of Reps et al. [20]. The IFDS algorithm requires an analysis whose domain is $\mathcal{P}(D)$ for some finite set $D$, and whose transfer functions are distributive. The analysis from the preceding sections satisfies these conditions; in this case, $D$ is the set of all possible alias sets (i.e., sets of local variables). IFDS is an efficient dynamic programming algorithm that evaluates the transfer functions on each individual alias set at a time, rather than on the set of all alias sets at a program point. The algorithm successively composes transfer functions for individual statements into transfer functions summarizing the effects of longer paths within a procedure. Once the composed transfer function summarizes all paths from the beginning to the end of a procedure, it can be substituted for any calls of the procedure.

Extending the IFDS algorithm to work on SSA form required one straightforward modification. The original algorithm tabulates the incoming dataflow set for each statement (i.e., the join of the outgoing dataflow sets of its predecessors). However, our more precise treatment of $\phi$ nodes requires processing the incoming flow set from each predecessor separately and joining the results only after the transfer function has been applied. Thus, we modified IFDS so that, for $\phi$ instructions only, it keeps track of a separate incoming dataflow set for each predecessor, instead of a single, joined incoming set.

We implemented the analysis within our framework for verifying temporal properties of multiple interacting objects [16,17]. The overall framework first performs an alias set analysis, then uses its results in a second analysis that tracks the state of groups of objects with respect to a given temporal safety property. Since the optimizations in this paper affect only the efficiency of the alias set analysis but not its results, the output of the client analysis is the same regardless of which implementation of the alias set analysis is used.

For empirical evaluation of the analysis we used the DaCapo Benchmark suite, version 2006-10-MR2 [2]. To deal with reflective class loading we instrumented the benchmarks using ProBe [15] and *J [7] to record actual uses of reflection at run time and provided the resulting reflection summary to the static analysis. The jython benchmark generates code at run time which it then executes; for this benchmark, we made the unsound assumption that the generated code does not call back into the original code and does not return any objects to it. We used the standard library from JDK 1.3.1_12 for antlr, pmd and bloat, and JDK 1.4.2_11 for the rest of the benchmarks, since they use features not present in JDK 1.3. We used the Soot framework [25] as a front-end to construct the intermediate representation that is the input to our analysis. We excluded the eclipse benchmark from our study because we were unable to make Soot soundly model the many uses of reflection in this benchmark. To give an indication of the sizes of the benchmarks, Figure 8 shows, for each benchmark, the number of methods reachable in the static call graph and the total number of nodes in the control flow graphs of the reachable methods.

We experimented with three different setups. Setup 1 used the default Set implementation of the Scala programming language. The sets are "immutable" in the sense that an update returns a new set object rather than modifying the existing set object. Usually, the implementations of the original and updated set share some of their data. The standard library provides customized implementations for sets of size 0 to 4 elements. For larger sets, a hash table implementation is used. According to the Scala documentation [18], the hash table-based implementation is optimized for sequential accesses where the last updated table is accessed most often. Accessing previous versions of the set is also made efficient by keeping a change log that is regularly compacted. In setup 2, the TreeSet data structure from the Scala API was used. This implementation uses balanced trees to store the set. An updated set reuses subtrees from the representation of the original set. Both setup 1 and 2 compute the alias sets on a program in non-SSA form and use the transfer functions from Figure 1. We also tried to apply setups 1 and 2 to programs in SSA form, but found them to run slower and use more memory than on the original, non-SSA IR. The third setup used the sorted list data structure with hash consing proposed in this paper. The analysis is computed on a program in SSA form and uses the transfer functions from Figure 7.

| Benchmark | Methods | CFG Nodes | SSA CFG Nodes |
|---|---|---|---|
| antlr | 4452 | 89442 | 96225 |
| bloat | 5955 | 95586 | 101179 |
| chart | 14912 | 241208 | 256367 |
| fop | 27408 | 410460 | 433249 |
| hsqldb | 11418 | 184201 | 198125 |
| jython | 14437 | 221215 | 234469 |
| luindex | 7358 | 113815 | 122447 |
| lusearch | 7821 | 114822 | 123674 |
| pmd | 9344 | 148101 | 155284 |
| xalan | 14961 | 227510 | 242788 |

**Figure 8.** Benchmark sizes: Column 2 gives the number of reachable methods for each benchmarks. Columns 3 and 4 give the total number of nodes in the control flow graphs (CFGs) of the reachable methods for each benchmark in non-SSA and SSA form respectively.

The following sections present the time and memory requirements of the three setups.

### 5.1 Running Time

Figure 9 compares the running times for the three setups; the white, grey and black bars represent running times for the first, second and third setup, respectively.

In all cases but antlr, the Set-based representation runs faster than the TreeSet-based representation. The maximum performance difference is in the case of luindex: the Set-based representation runs in 58% less time. On average (geometric mean), the Set-based representation runs in 28% less time than the TreeSet-based representation.

We compare the Set-based representation to our HCList representation. In all cases the HCList abstraction is faster. On average, the HCList representation runs in 58% less time than the Set-based representation. The largest speedup is achieved on the bloat benchmark, on which the HCList representation runs in 72% less time than the Set-based representation.

Although the conversion to SSA form increased the size of control flow graphs by 6.6% on average (Figure 8), the analysis is faster even on the larger control flow graphs.

### 5.2 Memory Consumption

Figure 10 shows the memory consumed by the different setups while computing the object abstraction. The reported memory use includes the memory required by the interprocedural object analysis, but excludes memory needed to store the intermediate representation and the control flow graph.

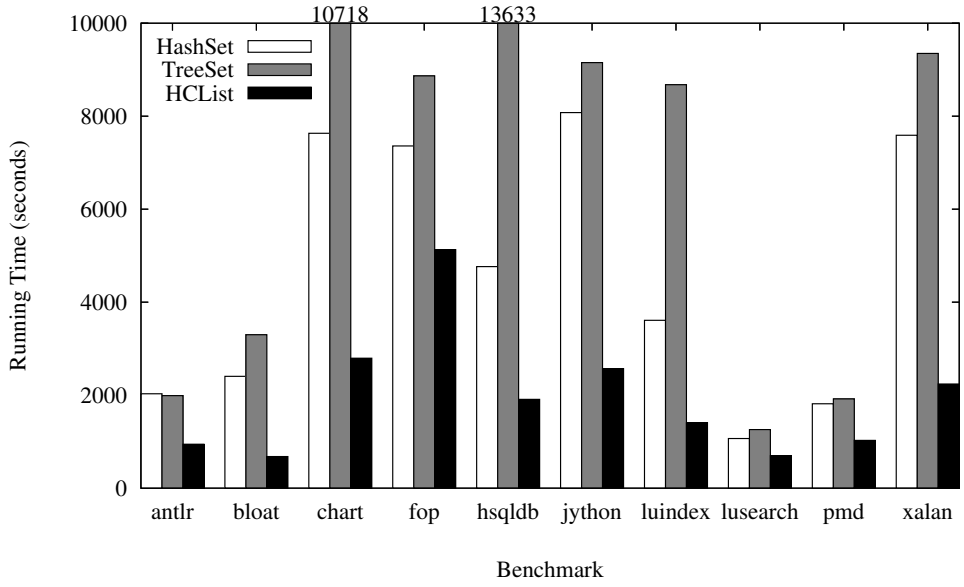


**Figure 9.** Running time for different data structures used in computing alias sets.

In all cases the `Set`-based representation uses less memory than the `TreeSet`-based representation of alias sets; the average reduction is 18% with the maximum of 40% for `hsqldb`. The `HCList` representation with hash consing uses even less memory than the `Set`-based representation. The average reduction is 44% and the maximum reduction is 74% in the case of `xalan`.

Even though the alias sets in the `HCList`-based representation may contain more variables than in the `Set` or `TreeSet`-based representation, the `HCList`-based representation requires less memory thanks to sharing of common tails of the linked lists.

## 6. Related Work

### 6.1 Alias Sets and Related Heap Abstractions

Jonkers [13] presented a storeless semantic model for dynamically allocated data. He noticed that in the store-based heap model that maps pointer variables to abstract locations, the abstract locations do not represent any meaningful information. Instead he defined an equivalence relation on the set of all heap paths. Deutsch [6] presented a storeless semantics of an imperative fragment of Standard ML. He used a right-regular equivalence relation on access paths to express aliasing properties of data structures. Alias sets are a special case of access path sets: they are sets of access paths with zero dereferences. Whereas an access path begins at a local variable and specifies a list of fields to be followed to reach the object, an alias set contains only the local variables pointing directly to the object.

Our inspiration to use sets of variables to represent abstract objects comes from the work of Sagiv et. al. [23]. This work presents a shape analysis that can be used to determine properties of heap-allocated data structures. For example, if the input to a program is a list (respectively, tree), is the output still a list (respectively, tree)? The shape analysis creates a shape graph in which each node is the set of variables pointing to an object. Pointer relationships between objects are represented by edges between the nodes. The graph is annotated with additional information; a predicate is associated with each node which indicates whether the particular node (abstract object) might be the target of multiple pointers emanating from different abstract objects. This is crucial for distinguishing between cyclic and acyclic data structures. Later work of Sagiv et al. [24] generalizes this idea by allowing the analysis designer to separate objects according to domain-specific user-defined predicates. Because our analysis computes the nodes of Sagiv's shape graph, it is possible to extend our analysis to Sagiv's analysis by keeping track of edges between the nodes. The SSA properties that we exploited and the ordered data structure that we employ can also be used in the shape analysis algorithm.

Hackett and Rugina [11] use a two layered heap abstraction to perform shape analysis that is scalable to large C programs. The first abstraction uses a flow-insensitive context-sensitive analysis to break the heap into chunks of disjoint memory locations called regions. Many regions are single variables; other regions represent areas of the heap. The second abstraction builds on top of the region-based memory partition, breaking the heap into small independent *configurations*. Each configuration represents a single heap location and keeps track of reference counts from other regions that target this location. Also, each configuration (abstract object) contains field access paths known to definitely reach (hit) or definitely not reach (miss) the object. Since in typical cases each region is a local variable the abstraction provides the same information as Sagiv's abstraction. Orlovich and Rugina [19] apply the analysis to detect memory leaks in C programs. Cherem and Rugina [3] adapt the abstraction to Java to perform compile-time deallocation of objects i.e. freeing the memory consumed by an object as soon as all references to it are lost. They use *configurations* to represent abstract objects and implement an efficient abstraction in the form of a *Tracked Object Structure* (TOS). A TOS maintains a compact representation of equivalent expressions making modifications to the heap abstraction efficient since each node in the data structure is an equivalence class. The efficiency of the abstraction could be further improved by maintaining the equivalence class representing the set of local variables that point to a particular concrete object as a sorted list using the total order imposed by a preorder traversal of the dominance tree.

In their work on typestate verification, Fink et. al. [8, 9] use a staged verifier to prove safety properties of objects. The most precise of these verifiers keeps track of which local variables must and must not point to the object along with similar information regarding incoming pointers (access paths) from other objects that must or must-not point to the object. Information about the allocation

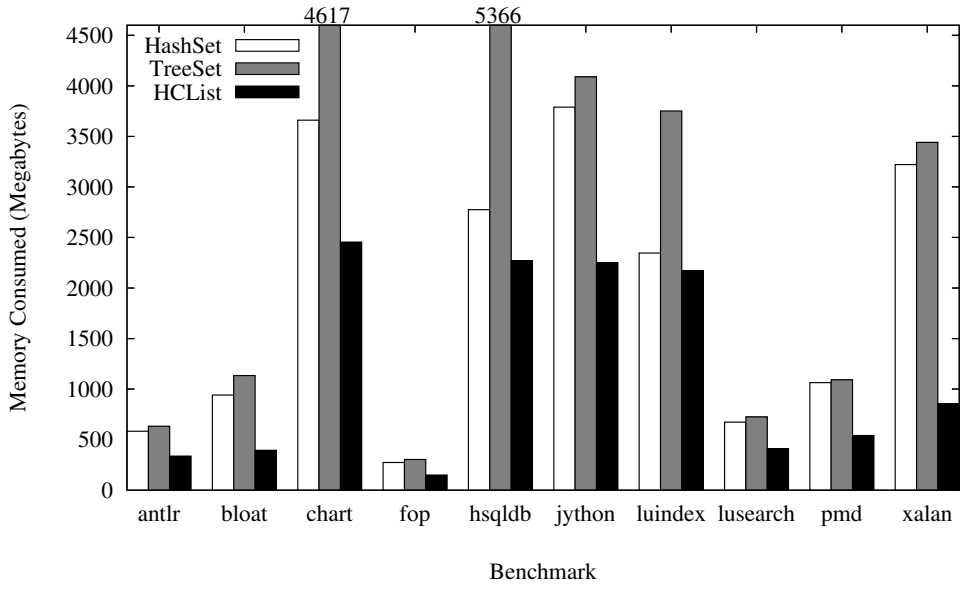


**Figure 10.** Memory consumed by different data structures used in computing alias sets.

site of the object is also maintained. This information is used to perform strong updates in the case when it can be proved that the points-to set of a receiver contains a single abstract object and that this single abstract object represents a single concrete object.

In earlier work [16, 17], we extend static typestate verification techniques for single objects to verify temporal specifications of multiple interacting objects. Whereas typestate verification typically associates a state with each abstract object, this is not possible when dealing with a state associated with multiple objects. We define two abstractions: a storeless heap abstraction based on alias sets and a second abstraction that associates a state with groups of related abstract objects.

A common technique used to precisely handle uncertainty due to heap loads is that of *materialization* or *focus* [3, 8, 9, 11, 17, 23]. Focus is important to regain the precision lost when an object is no longer referenced from any local variables, in which case the alias set analysis lumps it together with all other such objects. Focus splits the abstract object representation into two alias sets, one representing the single concrete object that was loaded, and the other representing all other objects previously represented by the alias set. The transfer functions in Figure 1 use focus for a heap load ($v \leftarrow \mathbf{e}$) by splitting $o^\sharp$ into two alias sets $o^\sharp \setminus \{v\}$ and $o^\sharp \cup \{v\}$. The focus operation in the transfer functions of Figures 6 and 7 no longer requires removing the variable $v$ from the resulting alias sets. As discussed in Section 4.1, the set difference operation is redundant in SSA form, since the original alias set $o^\sharp$ is guaranteed to not contain $v$.

### 6.2 Static Single Assignment (SSA) Form

Static Single Assignment form [1, 26] has been used as an intermediate representation since the late 1980s. Rosen et. al. [21] took advantage of SSA form to define a global value numbering algorithm. Cytron et al. [5] developed the now-standard efficient algorithm for converting programs to SSA form using dominance and dominance frontiers.

Hack et. al. [10] showed that the interference graph for register allocation of a program in SSA form is always chordal (i.e., its chromatic number equals the size of the largest clique). Such graphs can be optimally colored in quadratic time. The chordality of the interference graph is due to the SSA property that if the variables in some set $S$ are simultaneously live at some program point $p$, then they are all totally ordered by dominance, they are all live at the definition of the variable $v \in S$ dominated by all the others, and on every control flow path ending at $p$, the variable from $S$ defined last is $v$. Thus any relationship that holds between the variables at $p$ already holds at the definition of $v$. The abstraction presented in this paper is intuitively based on the same idea. Suppose the set of variables pointing to some concrete object $o$ at program point $p$ is $S$. Then those variables are totally ordered by dominance, and they all already pointed to $o$ when the variable in $v \in S$ dominated by the others was last defined. Thus if $S$ is represented by a linked list ordered by dominance, the transfer function for the instruction defining $v$ needs only to add $v$ to the head of the list. The only place where variables need to be removed from $S$ is an edge leading to a node no longer dominated by the definitions of those variables.

## 7. Conclusion

This paper focused on the core abstraction of an alias set used by numerous static analyses to infer properties about the pointers created and manipulated in a program. We presented a data structure implementing the alias set abstraction for programs in SSA form. The data structure consists of linked lists ordered by the preorder numbering of the dominance tree of the procedure. We showed that with this ordering, the transfer functions only apply local updates to the head of each list. Since the lists are ordered, common tails of different lists representing different alias sets can be shared. We implemented an interprocedural context-sensitive analysis using this representation of the abstraction. Our experimental results show that the ordered list representation is faster and requires less memory than standard set data structures. Running time decreased by 58% on average and by as much as 72% on one of the benchmarks. Memory requirements decreased by 44% on average, and by as much as 74% on one of the benchmarks.

## Acknowledgements

We thank the anonymous reviewers for their suggestions for improving the paper. This research was supported by the Natural Sciences and Engineering Research Council of Canada.

## References

[1] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–11. ACM Press, 1988.

[2] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. *OOPSLA '06: Proceedings of Object-Oriented Programming, Systems, Languages and Applications*, 2006.

[3] S. Cherem and R. Rugina. Compile-time deallocation of individual objects. *ISMM 06' Proceedings of the 2006 International Symposium on Memory Management*, pages 138–149, 2006.

[4] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1989.

[5] 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 Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.

[6] A. Deutsch. A storeless model of aliasing and its abstractions using finite representations of right-regular equivalence relations. *4th International Conference on Computer Languages*, pages 2–13, 1992.

[7] B. Dufour. Objective quantification of program behaviour using dynamic metrics. Master's thesis, McGill University, June 2004.

[8] S. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. *ISSTA '06: Proceedings of International Symposium on Software Testing and Analysis*, pages 133–144, 2006.

[9] S. J. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. *ACM Trans. Softw. Eng. Methodol.*, 17(2):1–34, 2008.

[10] S. Hack and G. Goos. Optimal register allocation for SSA-form programs in polynomial time. *Inf. Process. Lett.*, 98(4):150–155, 2006.

[11] B. Hackett and R. Rugina. Region-based shape analysis with tracked locations. *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 310–323, 2005.

[12] M. Hind. Pointer analysis: haven't we solved this problem yet? *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 54–61. ACM Press, 2001.

[13] H. B. M. Jonkers. Abstract storage structures. de Bakker and van Vliet, editors, *Algorithmic Languages*, pages 321–343. IFIP, North Holland, 1981.

[14] J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Inf.*, 7:305–317, 1977.

[15] O. Lhoták. Comparing call graphs. *PASTE '07: Proceedings of 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 37–42, 2007.

[16] N. A. Naeem and O. Lhoták. Extending typestate analysis to multiple interacting objects. Technical Report CS-2008-04, David R. Cheriton School of Computer Science, University of Waterloo, 2008. `http://www.cs.uwaterloo.ca/research/tr/2008/CS-2008-04.pdf`.

[17] N. A. Naeem and O. Lhoták. Typestate-like analysis of multiple interacting objects. *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 347–366, New York, NY, USA, 2008. ACM.

[18] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. Artima Press, first edition, 2008.

[19] M. Orlovich and R. Rugina. Memory leak analysis by contradiction. K. Yi, editor, *Static Analysis, 13th International Symposium, SAS 2006, Seoul, Korea, August 29-31, 2006, Proceedings*, volume 4134 of *Lecture Notes in Computer Science*, pages 405–424. Springer, 2006.

[20] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 49–61, 1995.

[21] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1988.

[22] B. G. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. G. Hedin, editor, *Compiler Construction, 12th International Conference, CC 2003*, volume 2622 of *Lecture Notes in Computer Science*, pages 126–137. Springer, 2003.

[23] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50, Jan. 1998.

[24] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, May 2002.

[25] R. Vallée-Rai, E. Gagnon, L. J. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: is it feasible? *Compiler Construction, 9th International Conference (CC 2000)*, volume 1781 of *Lecture Notes in Computer Science*, pages 18–34, 2000.

[26] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, 13(2):181–210, 1991.