# An Efficient Storeless Heap Abstraction Using SSA Form [*]

Nomair A. Naeem Ondřej Lhoták
D. R. Cheriton School of Computer Science
University of Waterloo, Canada
{nanaeem,olhotak}@uwaterloo.ca

**Abstract**

Precise, flow-sensitive analyses of pointer relationships often use a storeless heap abstraction. In this model, an object is represented using some abstraction of the expressions that refer to it (i.e. access paths). Many analyses using such an abstraction are difficult to scale due to the size of the abstraction and due to flow sensitivity. Typically, an object is represented by the set of local variables pointing to it, together with additional predicates representing pointers from other objects. The focus of this paper is on the set of local variables, the core of any such abstraction. Taking advantage of certain properties of static single assignment (SSA) form, we propose an efficient data structure that allows much of the representation of an object 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.

## 1 Introduction

Many static analyses have been proposed to infer properties about the pointers created and manipulated in a program. Points-to analysis determines to which objects a pointer

[*]Technical Report CS-2008-22. This research was supported by the Natural Sciences and Engineering Research Council of Canada.

may point, alias analysis determines whether two pointers point to the same object, and shape analysis determines the structure of the pointer relationships between a collection of objects. 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.

All of these static analyses require some way of abstracting the possibly unboundedly many objects in the heap. One such abstraction is based on the storeless heap model [6, 12]. This model represents an object by its *access paths*, the expressions that can be used to find the object in memory. An access path begins with a local variable followed by a sequence of field dereferences. In general, multiple access paths may reach the same object. Thus the abstraction represents each object by the set of access paths that reach it.

The storeless heap abstraction has been used in many analyses, especially shape analyses. Sagiv et al. [20] define an abstraction in which each concrete object is represented by the set of local variables that point to it. Thus, each abstract object is a set of variables. A key feature of this abstraction is that each abstract object (except the empty set of variables) corresponds to at most one concrete run-time object; this makes the abstraction precise and enables strong updates. On top of this abstraction of objects, the analysis maintains a set of edges between abstract objects representing the pointer relationships among corresponding concrete objects. Sagiv et al. further refine the object abstraction by allowing the analysis designer to separate objects according to domain-specific user-defined predicates [21]. Hackett and Rugina [10] define a related abstraction for C programs. Each abstract object contains a reference count from each "region", along with field access paths known to definitely reach (hit) or definitely not reach (miss) the object. Typically, each local variable is a region, so the reference counts in the abstraction provide the same information as Sagiv's abstraction. Orlovich and Rugina [17] apply the analysis to detect memory leaks in C programs. Cherem and Rugina [3] adapt the abstraction to Java. From the Java version of the abstraction, it is possible to determine the set of local variables pointing to the object. Fink et al. [8] define an abstraction that keeps track of which local variables must and must not point to the object, along with information about the allocation site of the object and incoming pointers from other objects. In previous work [14, 15] we have used a similar abstraction for typestate verification of multiple objects. We discuss some of these approaches in more detail in the Related Work section.

A common characteristic of all of these abstractions is that they are based on the set of local variables pointing to the object. This core abstraction is refined in a different way in each of these abstractions. Our contribution is an efficient representation of the set of local variables pointing to the object. This representation could be used as the core of an efficient implementation of each of these refined abstractions.

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 [9]. We exploit these same properties to efficiently represent the set of variables pointing to an object.

Analyses using the set-of-variables abstraction 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 using the abstraction are flow-sensitive, so many different variable 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. [20]. It is the second issue that is addressed by our work. 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 a set-of-variables object abstraction for programs in SSA form. The abstraction can be implemented using any set data structure, including ordered lists. The abstraction can be used as is in a shape analysis, or further refined with 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 [18], 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 set-of-variables 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 mention terms used in the remainder of the paper. Section 4 presents a new data structure and corresponding transfer functions for representing abstract objects. The implementation of an interprocedural context-sensitive analysis, able to work on different object abstractions, is discussed in Section 5. Empirical results comparing the running times and memory consumption of the analysis using different data structures for the abstraction are presented in Section 6. We discuss related work in Section 8 and give concluding remarks in Section 9.

## 2 A Set-based Storeless Heap Abstraction

This section defines how objects are represented in the abstraction, and presents a transfer function to determine the set of abstract objects at each program point.

The overall abstraction $\rho^\sharp$ is a set of abstract objects. This abstract set is an over-approximation of run-time behaviour. For every concrete object that could exist at run time at a given program point, the abstraction always contains an abstract object that abstracts that concrete object; however, the abstraction may contain additional abstract objects that do not correspond to any concrete object. Each abstract object $o^\sharp$ is a set of local variables of pointer type. The abstract object contains exactly those variables that point to the corresponding concrete object at run time. The set of variables in the abstract object 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 abstract objects, 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 $\{\{x\},\{x,z\},\{y\}\}$.

When the set of pointers in an abstract object is non-empty, the abstract object represents at most one concrete object at any given instant at run time. For example, consider the abstract object $\{x\}$. At run time, the pointer $x$ can only point to one concrete object $o$ at a time; thus at that instant, the abstract object $\{x\}$ represents only $o$ and no other concrete objects. This property enables very precise transfer functions for individual abstract objects, with strong updates. Continuing the example, the program statement `y := x` transforms the abstract object $\{x\}$ 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 abstract objects 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. [20] 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 abstract objects that further limit the concrete objects that they represent. For example, an abstract object 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.

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:

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

The constant **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 a set of transfer functions that specify the effect of an instruction on a single abstract object 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.

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

Figure 1: Transfer functions on individual abstract objects. 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 abstract objects 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 heap abstraction flow analysis is a forward dataflow analysis where the elements of the lattice are the abstract environments, $\rho^\sharp$. The merge operation is set union.

**Algorithm 1**: Dataflow Analysis

```
for each statement s, initialize out[s] to $\emptyset$
add all statements to worklist
while worklist not empty do
    remove some $s$ from worklist
    in = $\bigcup_{p \in \text{pred}(s)}$ out[$p$]
    out[s] = $[\![s]\!]_{\rho^\sharp}$( in )
    if out[s] has changed then
        foreach $s' \in succs(s)$ 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 dominate 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$(s)$).

## 4 Efficient storeless Heap Abstraction

To extend the transfer function from Figure 1 to SSA form, we define it for $\phi$ instructions in Figure 2. There is one important difference in the way that the transfer function for a $\phi$ instruction is evaluated, compared to the transfer functions for all other kinds of instructions. For instructions other than $\phi$ instructions, the analysis first computes the join (i.e. set union) of the dataflow facts on all incoming control flow edges, then applies the transfer function to the join. However, the effect of a $\phi$ instruction depends on which incoming control flow edge is used to reach it.

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

Figure 2: Transfer function for the $\phi$ instruction

Therefore, the transfer function for $\phi$ instructions shown in Figure 2 is dependent on an additional parameter, the control flow predecessor $p$. The transfer function first determines the parallel assignment $\sigma(\phi, p)$ that corresponds to the given incoming control flow edge $p$. The abstract object is then updated by adding all destination variables whose values are being assigned from variables already in the abstract object, and removing all variables whose values are being assigned from variables *not* in the abstract object. 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 $v_1 \leftarrow v_2$. Rather than first computing the join over all incoming control flow edges, the $\phi$ transfer function is computed separately for each incoming edge, and the join is computed *after* the $\phi$ instruction, on the results of the transfer function. This is more precise and corresponds more closely to the semantics of the $\phi$ instruction. Since the effect of a $\phi$ instruction depends on which control flow edge is used to reach the instruction, the abstract effect should be computed separately for each incoming edge, before the edge merges with the others. The dataflow analysis algorithm modified to process $\phi$ instructions in this way is shown in Algorithm 2.

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

```
for each statement s, initialize out[s] to $\emptyset$
add all statements to worklist
while worklist not empty do
    remove some $s$ from worklist
    if $s$ is a $\phi$ instruction then
        foreach $p \in preds(s)$ do
            out[$s$] = out[$s$] $\cup$ $[\![\phi]\!]_{\rho^\sharp}$(out[$p$], $p$)
    else
        in = $\bigcup_{p \in \mathrm{pred}(s)}$ out[$p$]
        out[s] = $[\![s]\!]_{\rho^\sharp}$( in )
    end
    if out[s] has changed then
        foreach $s' \in$ succs(s) do
            add $s'$ to worklist
    end
end
```

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 abstraction for objects 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 object 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 object abstraction. This reduces the size of each variable 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 object analysis, then intersect each abstract object computed by the transfer function with the set of live variables, as shown in the revised transfer function in Figure 3.

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

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

Figure 3: Transfer function with liveness filtering

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 vars(dom$(s)$), as shown in the modified transfer function in Figure 4.

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

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 abstract objects 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 abstract objects.

**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.

**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 out$[s]$ is a subset of vars(dom(s)). This is proved by induction

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

Figure 5: Simplified transfer function with dominance filtering

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 abstract objects 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:

$$\begin{array}{rl} & o^\sharp \setminus \{y_i : y_i \leftarrow x_i \in \sigma(\phi, p) \wedge x_i \notin o^\sharp\} \\ & \quad \cup \{y_i : y_i \leftarrow x_i \in \sigma(\phi, p) \wedge x_i \in o^\sharp\} \cap \text{vars}(\text{dom}(\phi)) \\ = & o^\sharp \setminus \text{defs}(\phi) \cup \{y_i : y_i \leftarrow x_i \in \sigma(\phi, p) \wedge x_i \in o^\sharp\} \cap (\text{defs}(\phi) \cup \text{vars}(\text{sdom}(\phi))) \\ = & o^\sharp \cap \text{vars}(\text{sdom}(\phi)) \cup \{y_i : y_i \leftarrow x_i \in \sigma(\phi, p) \wedge x_i \in o^\sharp\} \end{array}$$

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 abstract objects. The first operation is adding a variable defined in the current instruction to an abstract object. The second operation is intersecting each

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

Figure 6: Transfer functions without set difference operations

abstract object 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. We take advantage of the following property of the dominance tree.

**Property 2.** *Number the instructions in a procedure 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 to the 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 the variables in an arbitrary but consistent order. Parameters of the program, which are 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 $\texttt{prenum}(v_1) < \texttt{prenum}(v_2)$.

To represent each abstract object, 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 abstract objects that are a subset of vars(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 abstract object with vars(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 vars(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 $o^\sharp \cap$ vars(sdom($\phi$)) that occurs in the transfer function for a $\phi$ instruction. The incoming abstract object $o^\sharp$ 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 $\text{dom}(p) \setminus \text{sdom}(\phi)$. 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 abstract objects. Adding a variable to a set has been replaced by cons, and intersection with $\text{vars}(\text{sdom}(\phi))$ has been replaced by a call to prune.

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

Figure 7: Transfer functions on sorted lists

## 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** $\times$ `HCList` $\rightarrow$ `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. As an example consider the sequence of code shown on the left side of Figure 8. If each abstract object was represented separately as an unshared list of `ConsCells` then the four abstract objects at the end of the sequence would contain {a,b,d}, {a,b,d,e}, {a,b,c,d} and {a,b,c,d,e}, using a total of 16 `ConsCells`. However, with hash consing the same four abstract objects use only a total of 7 `ConsCells`.

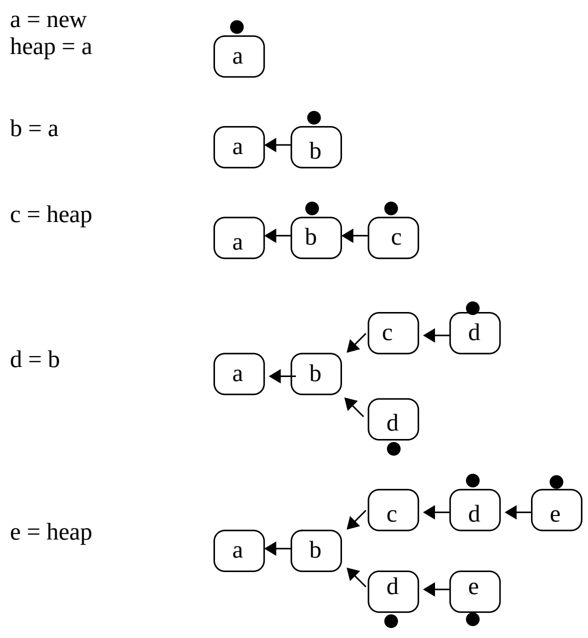

Figure 8: Sharing between different abstract objects. Filled circles represent the head of individual `HCLists`.

## 5 Interprocedural Analysis

The analysis defined in the preceding sections is intraprocedural. The analysis domain is $\mathcal{P}(\mathcal{P}(\textbf{Var}))$, where **Var** is the set of variables, and the merge operator is set union. The transfer functions are distributive. Thus, to extend the analysis to a context-sensitive interprocedural analysis, a natural choice is the interprocedural finite distributive subset (IFDS) algorithm of Reps et al. [18] with some small modifications which we explain in this section.

IFDS is a dynamic programming algorithm that uses $O(E|O^\sharp)|^3)$ time in the worst case, where $O^\sharp$ is the set of all possible abstract objects. The algorithm evaluates the transfer functions on each individual abstract object at a time, rather than on the set of all abstract objects at a program point. Thus, the algorithm uses the transfer functions for a single abstract object rather than the overall transfer function (i.e. $[\![s]\!]_{o^\sharp}$

rather than $[\![s]\!]_{\rho^\sharp}$). 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. Specifically, the algorithm uses a worklist to complete two tables of transfer functions: the PathEdge table gives the transfer function from the start node of each procedure to every other node in the same procedure, and the SummaryEdge table gives the transfer function that summarizes the effect of each call site in the program.

Extending the IFDS algorithm to work on SSA form required one straightforward modification. The PathEdge table in the original algorithm tracks the input flow set for each statement (i.e. the join of the output 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 the PathEdge table so that, for $\phi$ instructions only, it keeps track of a separate input set for each predecessor, instead of a single, joined input set.

The transfer functions $[\![s]\!]^6_{o^\sharp}$ and $[\![\phi]\!]^6_{o^\sharp}$ from Figure 7 can be used directly in the IFDS algorithm. In addition, we must also specify how to map abstract objects at a call site from the caller to the callee and back. The mapping into the callee is simple: for each abstract object, determine which of the actual arguments it contains, and create a new abstract object containing the corresponding formal parameters. We take care to keep the formal parameters in each of these newly created abstract object in the prenumber order defined for the callee.

In order to map objects from the callee back to the caller, a small modification to the IFDS algorithm is necessary. In the original algorithm, the return flow function is defined only in terms of the flow facts computed for the end node of the callee. In the callee, each abstract object is a set of variables of the callee, and it is not known which caller variables point to the object. However, the only place where the algorithm uses the return flow function is when computing a SummaryEdge flow function for a given call site by composing *return* $\circ$ $[\![p]\!]$ $\circ$ *call*, where *call* is the call flow function, $[\![p]\!]$ is the summarized flow function of the callee, and *return* is the return flow function. The original formulation of the algorithm assumes a fixed return flow function *return* for each call site. It is straightforward to modify the algorithm to instead use a function that, given a call site and the computed flow function $[\![p]\!]$ $\circ$ *call*, directly constructs the SummaryEdge flow function. A similar modification is also used in the typestate analysis of Fink et al. [8]. Indeed, the general modification is likely to be useful in other instantiations of the IFDS algorithm.

In the modified algorithm, the return flow function takes two arguments $o^\sharp_c$ and $o^\sharp_r$. The argument $o^\sharp_c$ is the caller-side abstraction of an object, the argument $o^\sharp_r$ is one possible callee-side abstraction of the same object at the return site, and the return flow function ought to yield the set of possible caller-side abstractions of the object after the call. Intuitively, after the call, the object is still pointed by the variables in $o^\sharp_c$, and may additionally be pointed to by the variable at the call site to which the result of the call is assigned, provided the callee-side variable being returned is in the callee-side abstraction of the object. We write $v_s$ to denote the callee-side variable being returned and $v_t$ to denote the caller-side variable to which the result of the call is assigned.

Formally, the return function is defined as follows.

$$\text{ret}(o_c^\sharp, o_r^\sharp) \triangleq \begin{cases} o_c^\sharp \cup \{v_t\} & \text{if } v_s \in o_r^\sharp \\ o_c^\sharp & \text{otherwise} \end{cases}$$

Like the intraprocedural transfer functions, the return function only adds the variable defined at the call site to an abstract object. Thus, like the intraprocedural transfer functions, the addition can be implemented using a simple cons operation on the ordered list. In the case of an object newly created within the callee that did not exist before the call, the empty set is substituted for $o_c^\sharp$, since no variables of the caller pointed to the object before the call.

## 5.1 Number of Abstract Objects

As Sagiv et al. point out ( [20, Section 6.2]) the number of possible abstract objects is bounded by $2^{\textbf{Var}}$. They indicate that it is possible to use *widening* to eliminate the possibility of an exponential blowup. We modify the IFDS algorithm to widen whenever the number of abstract objects increases beyond a set threshold. When this happens, we widen by coalescing different abstract objects by discarding some, already computed, precise information. Specifically, we coalesce abstract objects by choosing some variable $v$, and forgetting whether or not any abstract object contains $v$. As an example consider two abstract objects {a} and {a,b}. The second abstract object indicates that both variables $a$ and $b$ point to the concrete object. We widen these abstract objects by discarding the precise knowledge about $b$ and considering that any abstract object may or may not include $b$. This transforms the abstract objects to : {a!,b?} and {a!,b?} where a! means that the abstract object definitely includes a, and b? means that the abstract object may or may not contain b. Since the two abstract objects have become identical, they are merged into one, thereby reducing the overall number of abstract objects.

An important question is how to choose the variable for widening. We experimented with two possible heuristics:

1. Choose the variable with the lowest preorder number, since it is the least recently defined and may no longer be live.
2. Choose the most recently added variable, since it is likely to have caused a large blowup.

Although a more thorough experimental evaluation is needed, in our preliminary experiments, the second heuristic tended to a lower overall number of widenings. Therefore, we have used this heuristic for all of the experiments reported in the following section.

# 6 Empirical Evaluations

For empirical evaluation of the analysis we used a subset of the DaCapo benchmark suite, version 2006-10-MR2 [2] for our experiments (antlr, bloat, pmd, jython, hsqldb, luindex, xalan and chart). To deal with reflective class loading we instrumented the benchmarks using ProBe [13] 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 1.3. To give an indication of the sizes of the benchmarks, Figure 9 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.

| Benchmark | Methods | CFG Nodes | SSA CFG Nodes |
|---|---|---|---|
| antlr | 4452 | 89437 | 96227 |
| bloat | 5955 | 95588 | 101177 |
| pmd | 9344 | 148103 | 155292 |
| jython | 14437 | 221217 | 234458 |
| hsqldb | 11418 | 184196 | 198134 |
| luindex | 7358 | 113810 | 122450 |
| xalan | 14961 | 227504 | 242785 |
| chart | 14912 | 241216 | 256348 |

Figure 9: 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.

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` API specification [16], the hash table-based implementation is optimized for sequential accesses where the last updated table is accessed most often. Accessing previous version 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 heap abstraction 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.

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

## 6.1 Running Time

Figure 10 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.

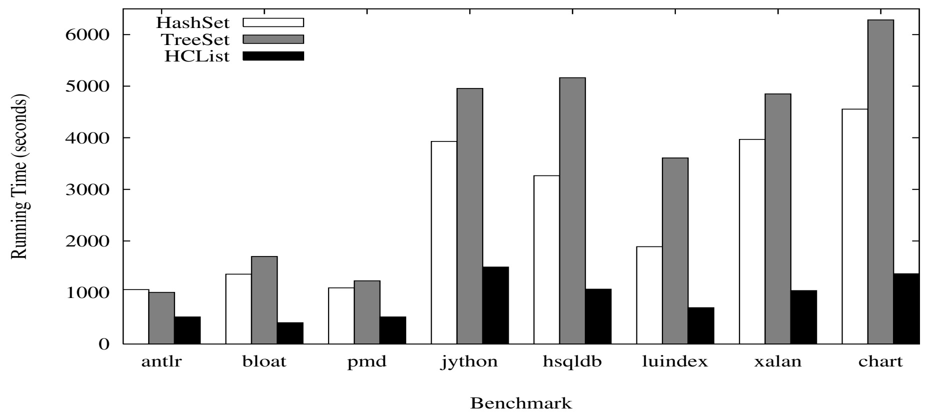

Figure 10: Running time for different data structures used in computing the heap abstraction.

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 is 48% faster. On average, the `Set`-based representation is 22% faster than the `TreeSet`-based representation.

We compare the `Set`-based representation to our `HCList` representation. In all cases the `HCList` abstraction is faster. The average running time improvement is 63%, and the maximum is 74% on the `xalan` benchmark.

Although the conversion to SSA form increased the size of control flow graphs by 6.5% on average (Figure 9), the analysis is faster even on the larger control flow graphs.

## 6.2 Memory Consumption

Figure 11 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.

In all cases the `Set`-based representation uses less memory than the `TreeSet`-based representation of abstract objects; the average reduction is 12%. The `HCList` representation with hash consing uses even less memory than the `Set`-based representation. The average reduction is 43% and the maximum reduction is 68% in the case of `xalan`.

Even though the abstract objects 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.

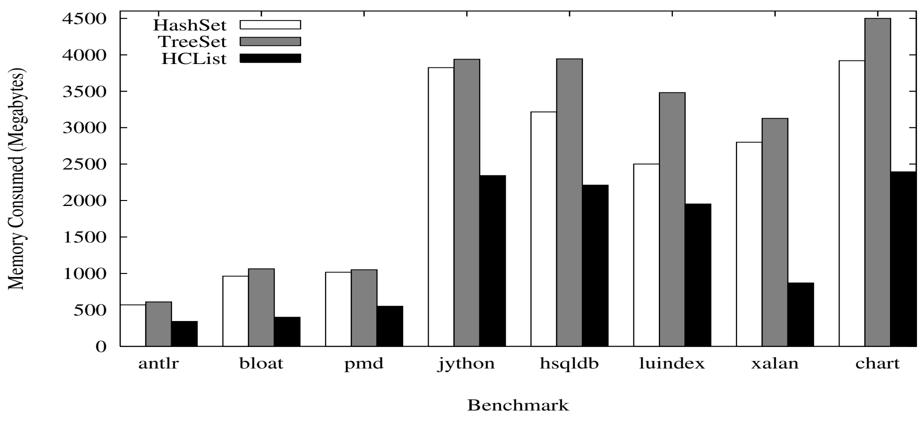

Figure 11: Memory consumed by different data structures used in computing the heap abstraction.

## 7 Future Work

In this section we discuss some future work for the storeless heap abstraction presented in this paper.

### 7.1 Implementation

This section looks at ways in which the performance of the presented abstraction could be further improved. We also discuss some additional experiments that we are interested in performing:

**Profiling:**
Due to time constraints we were unable to profile our implementation both for time and memory usage. Since the goal of this work is computing an efficient storeless heap abstraction, it is essential we optimize the code by detecting performance bottlenecks. For run time performance, although we were careful in designing the operations performed during the application of the transfer functions, profiling information might suggest additional avenues for speed improvements. It might, however, turn out that the cost of computing the abstraction is dominated by the number of abstract objects that need to be processed. While we have tried to keep this number as small as possible, by producing as few abstract objects as possible and merging similar abstract objects,

there is not much that can be done to reduce the number further. One optimization, that has been implemented, though not discussed in the paper, is to maintain a set $h^\sharp$, a subset of the set $\rho^\sharp$. Abstract objects that have escaped to the heap, via store statements, are added to this subset. The transfer function for loads ($v \leftarrow e$), is then modified to apply the focus operation only for abstract objects that are in the escaped set of objects ($o^\sharp \in h^\sharp$). Adding this extra condition restricts the number of focus operations to those abstract objects which have escaped to the heap and hence reduces the overall number of abstract objects created.

We also intend to profile the implementation for memory usage to see if there are any places where memory consumption could be reduced. One such possibility are the support data structures and specially the custom data structure representing the abstract objects. We discuss this next.

**HashConsing**

As mentioned in Section 4.3 we use Hash Consing to share the tails of the ordered lists representing abstract objects. However, even without memory profiling information, we noticed that we store a lot of redundant information in each `ConsCell`. Currently, each `ConsCell` in the implementation contains references for the method containing the variable, a reference to the variable itself, an integer value representing the pre-order number assigned to the variable, a unique identifier for the `ConsCell`, the size of the `HCList` starting at this `ConsCell` and a pointer to the next `ConsCell` in the list, if one exists. The method and variable are needed so that each variable can be uniquely identified in the program. However, for a given method, this results in a lot of redundant references to the method. It should be possible to simply use the pre-order number assigned to the variable to uniquely identify the variable. This will reduce the memory consumed by each `ConsCell`. Currently, we had decided to explicitly store the size of an `HCList` in the `size` field. This improves performance since the size does not need to be computed over and over again. A tradeoff would be to remove this field, hence reducing memory consumed, at the cost of needing to compute the size every time this value is required by the transfer functions. We intend to experiment with this tradeoff. Although, we have not yet performed any specific experiments to measure the usefulness of hash consing we intend to do so by using the simplified transfer functions with and without hash consing.

**Live Variables**

In Figure 6 we present the simplified transfer functions, using the liveness property and dominance filtering. However, our initial experiments using these transfer functions showed that the transfer functions performed poorly compared to the transfer functions which use live variable filtering. The reasoning for this is that although dominance filtering removes those variables from the abstract object whose definition does not dominate the current statement there are instances where a variable is no longer live but whose definition still dominates the statement i.e. the variables are in live-out($s$) but not in vars(dom($s$). These variables, although irrelevant, are not removed by dominance filtering. Although, we had hoped that this would not have a major effect on the size of the abstract object it turns out that even if the size does not increase drastically the number of abstract objects created does. This is so because abstract objects which only differ in the no longer live variables can not be merged to reduce the number of abstract objects. We intend to investigate whether it might be possible to apply dominance filtering and then live variable filtering for some but not all statements.

### 7.2 Client Analyses

Our evaluation computes the improvements in speed and memory consumption but does not validate the abstraction computed. Although, we are confident that the new abstraction is as precise as the original abstraction we intend to verify this by using the storeless heap abstraction in some client analyses. One such analysis is the tracematch analysis from our previous work on verifying temporal safety properties of multiple interacting objects [15]. This analysis requires an object abstraction in order to be able to ascertain relationships between multiple objects. By plugging in the new abstraction proposed in this work we can easily verify whether the abstraction is at least as precise as the set-based abstraction that the analysis currently uses.

It would also be interesting to investigate how easily the presented abstraction can be extended to the abstractions discussed in the related work section of this paper. As we mentioned earlier, the storeless heap abstraction we present is the core set-of-variables abstraction used by a number of static analyses inferring properties of pointers in the program. Extending our abstraction to perform these static analyses should lead to performance improvements for these analyses.

## 8 Related Work

### 8.1 Heap Abstraction

Jonkers [12] 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.

Our inspiration to use variables to represent abstract objects comes from the work of Sagiv et. al. [20]. 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. [21] 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 [10] 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 [17] 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] 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 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.

Our previous work on verification of multi-object temporal specifications [14, 15] extends static typestate verification techniques for single objects to multiple interacting objects. Whereas typestate verification typically associates a state to 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 sets-of-variables and a second abstraction which associates a state to groups of related abstract objects. Although in [14, 15] we used a `Set`-based representation for the storeless heap abstraction, we intend to take advantage of the data structure presented in this paper.

A common technique used to precisely handle uncertainty due to heap loads is that of *materialization* or *focus* [3, 8, 10, 15, 20]. Focus is important to regain the precision lost when an object is no longer referenced from any local variables, in which case the analysis lumps it together with all other such objects. Focus splits the abstract object representation into two, one representing the single concrete object that was loaded, and the other representing all other objects previously represented by the abstract object. The transfer functions in Figure 1 use focus for a heap load ($v \leftarrow \mathbf{e}$) by splitting $o^\sharp$ into two abstract objects $o^\sharp \setminus \{v\}$ 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 abstract objects. As discussed in Section 4.1, the set difference operation is redundant in SSA form, since the original abstract object $o^\sharp$ is guaranteed to not contain $v$.

## 8.2 Static Single Assignment (SSA) Form

Static Single Assignment form [1, 22] has been used as an intermediate representation since the late 1980s. Rosen et. al. [19] 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. [9] 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.

Hasti et. al in [11] propose an algorithm which can improve results of flow-insensitive points-to analysis by iteratively converting the program into SSA form and applying a flow-insensitive points-to analysis to it. After each iteration the points-to sets might shrink (become more precise) but are always guaranteed to be safe (a superset of the points-to sets given by a flow-sensitive analysis). They conjecture that reaching a fixed point of this iterative approach might lead to points-to results with similar precision as that of a flow-sensitive points-to analysis.

# 9 Conclusion

This paper focused on the core abstraction of a set of variables used by numerous static analyses inferring properties about the pointers created and manipulated in a program. We presented a data structure implementing the set-of-variables object 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 abstract objects 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. The speedup was 63% on average, and as high as 74% on one of the benchmarks. Memory requirements decreased by 43% on average, and as much as 68% on one of the benchmarks.

## 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. IEEE Computer Society Press, 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. Hack and G. Goos. Optimal register allocation for SSA-form programs in polynomial time. *Inf. Process. Lett.*, 98(4):150–155, 2006.

[10] 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.

[11] R. Hasti and S. Horwitz. Using static single assignment form to improve flow-insensitive pointer analysis. *PLDI '98: Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, pages 97–105, New York, NY, USA.

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

[13] 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.

[14] N. A. Naeem and O. Lhoták. Extending typestate analysis to multiple interacting objects. *OOPSLA '08: Proceedings of Object-Oriented Programming, Systems, Languages and Applications*.

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

[16] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala: A comprehensive step-by-step guide*. Preprint edition, 2008.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

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