Precise Null Pointer Analysis Through Global Value Numbering

Ankush Das[1] and Akash Lal[2]

[1] Carnegie Mellon University, Pittsburgh, PA, USA
[2] Microsoft Research, Bangalore, India

Abstract. Precise analysis of pointer information plays an important role in many static analysis tools. The precision, however, must be balanced against the scalability of the analysis. This paper focusses on improving the precision of standard context and flow insensitive alias analysis algorithms at a low scalability cost. In particular, we present a semantics-preserving program transformation that drastically improves the precision of existing analyses when deciding if a pointer can alias NULL. Our program transformation is based on Global Value Numbering, a scheme inspired from compiler optimization literature. It allows even a flow-insensitive analysis to make use of branch conditions such as checking if a pointer is NULL and gain precision. We perform experiments on real-world code and show that the transformation improves precision (in terms of the number of dereferences proved safe) from 86.56% to 98.05%, while incurring a small overhead in the running time.
Keywords: Alias Analysis, Global Value Numbering, Static Single Assignment, Null Pointer Analysis

1 Introduction

Detecting and eliminating null-pointer exceptions is an important step towards developing reliable systems. Static analysis tools that look for null-pointer exceptions typically employ techniques based on *alias analysis* to detect possible aliasing between pointers. Two pointer-valued variables are said to *alias* if they hold the same memory location during runtime. Statically, aliasing can be decided in two ways: (a) *may-alias* [1], where two pointers are said to may-alias if they can point to the same memory location under some possible execution, and (b) *must-alias* [27], where two pointers are said to must-alias if they always point to the same memory location under all possible executions. Because a precise alias analysis is undecidable [24] and even a flow-insensitive pointer analysis is NP-hard [14], much of the research in the area plays on the precision-efficiency trade-off of alias analysis. For example, practical algorithms for may-alias analysis lose precision (but retain soundness) by over-approximating: a verdict that two pointer may-alias does not imply that there is some execution in which they actually hold the same value. Whereas, a verdict that two pointers cannot may-alias must imply that there is no execution in which they hold the same value.

We use a sound may-alias analysis in an attempt to prove the safety of a program with respect to null-pointer exceptions. For each pointer dereference, we ask the analysis if the pointer can may-alias NULL just before the dereference. If the answer is that it cannot may-alias NULL, then the pointer cannot hold a NULL value under all possible executions, hence the dereference is safe. The more precise the analysis, the more dereferences it can prove safe. This paper demonstrates a technique that improves the precision of may-alias analysis at a little cost when answering aliasing queries of pointers with the NULL value.

The NULL value is special because programmers tend to be defensive against null-pointer exceptions. If there is doubt that a pointer, say x, can be NULL or not, the programmer uses a check "if $(x \neq NULL)$" before dereferencing x. Existing alias analysis techniques, especially flow insensitive techniques for may-alias analysis, ignore all branch conditions. As we demonstrate in this paper, exploiting these defensive checks can significantly increase the precision of alias analysis. Our technique is based around a semantics-preserving program transformation and requires only a minor change to the alias analysis algorithm itself.

Program transformations have been used previously to improve the precision for alias analysis. For instance, it is common to use a *Static Single Assignment* (SSA) conversion [5] before running flow-insensitive analyses. The use of SSA automatically adds some level of flow sensitivity to the analysis [12]. SSA, while useful, is still limited in the amount of precision that it adds, and in particular, it does not help with using branch conditions. We present a program transformation based on *Global Value Numbering* (GVN) [16] that adds significantly more precision on top of SSA by leveraging branch conditions.

The transformation works by first inserting an assignment $v := e$ on the *then* branch of a check if $(e \neq NULL)$, where v is a fresh program variable. This gives us the global invariant that v can never hold the NULL value. However, this invariant will be of no use unless the program uses v. Our transformation then searches locally, in the same procedure, for program expressions e' that are *equivalent* to e, that is, at runtime they both hold the same value. The transformation then replaces e' with v. The search for equivalent expressions is done by adapting the GVN algorithm (originally designed for compiler optimizations [10]).

Our transformation can be followed up with a standard alias analysis to infer the points-to set for each variable, with a slight change that the new variables introduced by our transformation (such as v above) cannot be NULL. This change stops spurious propagation of NULL and makes the analysis more precise. We perform extensive experiments on real-world code. The results show that the precision of the alias analysis (measured in terms of the number of pointer dereferences proved safe) goes from 86.56% to 98.05%. This work is used in Microsoft's Static Driver Verifier tool [22] for finding null-pointer bugs in device drivers[3].

The rest of the paper is organized as follows: Section 2 provides background on flow-insensitive alias analysis and how SSA can improve its precision. Section 3 illustrates our program transformation via an example and Section 4 presents

---

[3] `https://msdn.microsoft.com/en-us/library/windows/hardware/mt779102(v=vs.85).aspx`

```
var x :  int
procedure f(var y :  int) returns u :  int
{
    var z :  int
    L1 :
        x := y.f;
        assume (x ≠ Null);
        goto L2;
    L2 :
        z.g := y;
        assert (x ≠ Null);
        u := x;
        return;
}
```

```
procedure main()
{
    var a :  int;
    var b :  int;
    L1 :
        a := new();
        b := call f(a);
        goto L2;
    L2 :
        return;
}
```

Fig. 1. An example program in our language

it formally. Section 5 presents experimental results, Section 6 describes some of the related work in the area and Section 7 concludes.

2 Background

2.1 Programming Language

We introduce a simplistic language to demonstrate the alias analysis and how program transformations can be used to increase its precision. As is standard, we concern ourselves only with statements that manipulate pointers. All other statements are ignored (i.e., abstracted away) by the alias analysis. Our language has assignments with one of the following forms: pointer assignments x := y, dereferencing via field writes x.f := y and reads x := y.f, creating new memory locations x := new(), or assigning NULL as x := NULL. The language also has assume and assert statements:

- assume B checks the Boolean condition B and continues execution only if the condition evaluates to *true*. The assume statement is a convenient way of modeling branching in most existing source languages. For instance, a branch "if (B)" can be modeled using two basic blocks, one beginning with assume B and the other with assume ¬B.
- assert B checks the Boolean condition B and continues execution if it holds. If B does not hold, then it raises an assertion failure and stops program execution.

A program in our language begins with global variable declarations followed by one or more procedures. Each procedure starts with declarations of local variables, followed by a sequence of basic blocks. Each basic block starts with a label, followed by a list of statements, and ends with a control transfer, which is either a goto or a return. A goto in our language can take multiple labels. The choice between which label to jump is non-deterministic. Finally, we disallow loops in the control-flow of a procedure; they can instead be encoded using

Statement	Constraint
$i:$ x := new()	$aS_i \in pt(x)$
x := Null	$aS_0 \in pt(x)$
x := y	$pt(y) \subseteq pt(x)$
x := y.f	$aS_i \in pt(y)$ $pt(aS_i.\mathsf{f}) \subseteq pt(x)$
x.f := y	$aS_i \in pt(x)$ $pt(y) \subseteq pt(aS_i.\mathsf{f})$

Fig. 2. Program statements and corresponding points-to set constraints

---

Algorithm 1 Algorithm for computing points-to sets

1: For each program variable x, let $pt(x) = \emptyset$
2: **repeat**
3:    $opt := pt$
4:    **for all** program statements `st` **do**
5:       **if** `st` is $i:$ x := new() **then**
6:          $pt(x) := pt(x) \cup \{aS_i\}$
7:       **if** `st` is x := Null **then**
8:          $pt(x) := pt(x) \cup \{aS_0\}$
9:       **if** `st` is x := y **then**
10:        $pt(x) := pt(x) \cup pt(y)$
11:       **if** `st` is x := y.f **then**
12:          **for all** $aS_i \in pt(y)$ **do**
13:             $pt(x) := pt(x) \cup pt(aS_i.\mathsf{f})$
14:       **if** `st` is x.f := y **then**
15:          **for all** $aS_i \in pt(x)$ **do**
16:             $pt(aS_i.\mathsf{f}) := pt(aS_i.\mathsf{f}) \cup pt(y)$
17:       **for all** $tagged(x)$ **do**
18:          $pt(x) := pt(x) - \{aS_0\}$
19: **until** $opt = pt$

---

procedures with recursion. This restriction simplifies the presentation of our algorithms. Figure 1 shows an illustrative example in our language.

2.2   Alias Analysis

This section describes Andersen's may-alias analysis [1]. The analysis is context and flow-insensitive, which means that it completely abstracts away the control of the program. But the analysis is field-sensitive, which means that a value can be obtained by reading a field f only if it was previously written to the same field f. Field-insensitive analyses, for example, also abstract away the field name.

The analysis outputs an over-approximation of the set of memory locations each pointer can hold under all possible executions. Since a program can potentially execute indefinitely (because of loops or recursion), the number of memory locations allocated by a program can be unbounded. We consider a finite abstraction of memory locations, commonly called the *allocation-site abstraction* [15]. Each memory location allocated by the same new statement is represented using the same abstract value. This abstract value is also called an allocation site. We label each new statement with a unique number i and refer to its corresponding allocation site as aS_i. We use the special allocation site aS_0 to denote Null.

We follow a description of Andersen's analysis in terms of set constraints [26], shown in Figure 2. The analysis outputs a points-to relation pt where $pt(x)$ represents the points-to set of a variable x, i.e. (an over-approximation of) the

```
x := new();                          x1 := new();
assert (x ≠ Null);                   assert (x1 ≠ Null);
y := x.f;                            y := x1.f;
x := Null;                           x2 := Null;
```

Fig. 3. A program snippet before SSA (left) and after SSA (right)

set of allocation sites that x may hold under all possible executions. In addition, it also computes $pt(aS_i.\mathsf{f})$, for each allocation site aS_i and field f, representing (an over-approximation of) the set of values written to the f field of an object in $pt(aS_i)$.

The analysis abstracts away program control along with **assert** and **assume** statements. It considers a program as a bag of pointer-manipulating statements where each statement can be executed any number of times and in any order. Function calls are processed by adding assignments between formal and actual variables. For instance, in Figure 1, the call to f from **main** will result in the assignments y := a and b := u. For each statement, the analysis follows Figure 2 to generate a set of rules that define constraints on the points-to solution pt. The rules can be read as follows.

- If a program has an allocation x := new() and this statement is labeled with the unique integer i, then the solution must have $aS_i \in pt(x)$.
- If a program has the statement x := NULL, then it must be that $aS_0 \in pt(x)$.
- If the program has an assignment x := y then the solution must have $pt(y) \subseteq pt(x)$, because x may hold any value that y can hold.
- If the program has a statement x := y.f and $aS_i \in pt(y)$, then it follows that $pt(aS_i.\mathsf{f}) \subseteq pt(x)$ because x may hold any value written to the f field of aS_i.
- If the program has a statement x.f := y and $aS_i \in pt(x)$ then it must be that $pt(y) \subseteq pt(aS_i.\mathsf{f})$.

These set constraints can be solved using a simple fix-point iteration, shown in Algorithm 1. (Our tool uses a more efficient implementation [26].) For now, ignore the loop on line 17. Once the solution is computed, we check all assertions in the program. We say that an assertion **assert** $(x \neq \text{NULL})$ is *safe* (i.e., the assertion cannot be violated) if $aS_0 \notin pt(x)$. We do not consider other kinds of assertions in the program because our goal is just to show null-exception safety. Andersen's analysis complexity is cubic in the size of the program.

2.3 Static Single Assignment (SSA)

This section shows how a program transformation can improve the precision of an alias analysis. Consider the program on the left in Figure 3. A flow-insensitive analysis does not look at the order of statements. Under this abstraction, the analysis cannot prove the safety of the assertion in this snippet of code because it does not know that the assignment of NULL to x only happens after the assertion.

To avoid such loss in precision, most practical implementations of alias analysis use the Single Static Assignment (SSA) form [5]. Roughly, SSA introduces

```
assume (x ≠ Null);
y := x;
assert (x ≠ Null);
z := x.f;
```

```
assume (x ≠ Null);
cseTmp# := x;
y := cseTmp#;
assert (cseTmp# ≠ Null);
z := cseTmp#.g;
```

Fig. 4. A program snippet before CSE (left) and after CSE (right)

multiple copies of each original variable such that each variable in the new program only has a single assignment. The SSA form of the snippet is shown on the right in Figure 3. Clearly, this program has the same semantics as the original program. But a flow-insensitive analysis will now be able to show the safety of the assertion in the program because the assignment of NULL is to $x2$ whereas the assertion is on $x1$.

3 Overview

This section presents an overview of our technique of using program transformations that add even more precision to the alias analysis compared to the standard SSA. We start by using *Common Subexpression Elimination* [4] and build towards using *Global Value Numbering* [16], which is used in our implementation and experiments.

3.1 Common Subexpression Elimination

We demonstrate how we can leverage **assume** and **assert** statements to add precision to the analysis. Consider the program on the left in Figure 4. Once the program control passes the **assume** statement, we know that x cannot point to NULL, hence the assertion is safe, irrespective of what preceded this code snippet. Also, note that SSA renaming does not help prove the assertion in this case (it is essentially a no-op for the program). We now make the case for a different program transformation.

As a first step, we introduce a new local variable $\mathsf{cseTmp}^{\#}$ to the procedure and assign it the value of x right after the **assume**. These new variables that we introduce to the program will carry the tag "#" to distinguish them from other program variables. For a tagged variable $\mathsf{w}^{\#}$, we say that $tagged(\mathsf{w}^{\#})$ is *true*. These tagged variables carry the special invariant that they cannot be NULL; their only assignment will be after an assume statement that prunes away the NULL value. (The same reasoning applies to **assert** statements too, i.e. once control passes a statement $\mathsf{assert}(\mathsf{x} \neq \text{NULL})$, x cannot point to NULL.)

After introducing the variable $\mathsf{cseTmp}^{\#}$, we make use of a technique similar to *Common Subexpression Elimination* (CSE) to replace all expressions that are equivalent to $\mathsf{cseTmp}^{\#}$ with the variable itself, resulting in the program on the right in Figure 4. This snippet is clearly equivalent to the original one. We perform the alias analysis on this snippet as usual, but enforce that $pt(cseTmp^{\#})$ cannot have aS_0 because it cannot be NULL. (See the loop on line 17 of Algorithm 1.) The analysis can now prove that the assertion is safe.

The process of finding equivalent expressions is not trivial. For instance, consider the following program where we have introduced the variable cseTmp$^\#$.

```
assume (x.f ≠ Null);
cseTmp# := x.f;
y.f := z;
z := x.f;
```

In the last assignment, x.f cannot be substituted by cseTmp$^\#$, because there is an assignment to the field f in the previous statement. As there is no aliasing information present at this point, we have to conservatively assume that y and x could be aliases, thus, the assignment y.f := z can potentially change the value of x.f, breaking its equivalence to cseTmp$^\#$.

3.2 Global Value Numbering

We improve upon the previous transformation by using a more precise method of determining expression equalities. The methodology remains the same: we introduce temporary variables that cannot be NULL and use them to replace syntactically equivalent expressions. But this time we adapt the Global Value Numbering (GVN) scheme to detect equivalent expressions. Consider the following program. (For now, ignore the right-hand side of the figure after the "$\Longrightarrow$".)

```
1  y := x.f.g;          ⟹  t₁ ← x, t₂ ← t₁.f, t₃ ← t₂.g, y ↪ t₃
2  z := y.h;            ⟹  t₃ ← y, t₄ ← t₃.h, z ↪ t₄
3  assume (z ≠ Null);   ⟹  add t₄ to nonNullExprs
4  a := x.f;            ⟹  t₁ ← x, t₂ ← t₁.f, a ↪ t₂
5  b := a.g.h;          ⟹  t₂ ← a, t₃ ← t₂.g, t₄ ← t₃.h, b ↪ t₄
6  assert (b ≠ Null);   ⟹  check t₄ ∈ nonNullExprs
7  c.g := d;
```

It is clear that z and b are equivalent at the assertion location, and because $z \neq$ NULL, the assertion is safe. However, none of the previous methods would allow us to prove the safety of the assertion. We adapt the GVN scheme to help us establish the equality between z and b. We introduce the concept of *terms* that will be used as a placeholder for subexpressions. The intuitive idea is that equivalent subexpressions will be represented using the same term.

We start by giving an overview of the transformation for a single basic block, and then generalize it to full procedure later in this section. For a single basic block, we walk through the statements in order and as we encounter a new variable, we assign it a new term and remember this mapping in a dictionary called *hashValue*. We also store the mapping from terms to other terms through operators in a separate dictionary called *hashFunction*. For example, if x is assigned term t_1, and we encounter the assignment y := x.f, we store *hashFunction*[f]$[t_1] = t_2$ and assign the term t_2 to y. We also maintain a separate list *nonNullExprs* of terms that are not null. Finally, for performing the actual substitution, we maintain a dictionary *defaultVar* that maps terms to the temporary variables that we introduced for non-null expressions.

We go through the program snippet starting at the first statement and move down to the last statement. At statement i, we follow the description written in the ith item below. This description is also shown on the right side of the program snippet, after the $\Longrightarrow$ arrow.

1. Assign a new term t_1 to x, and set $hashValue[\mathsf{x}] = t_1$. Then, set $hashFunction[\mathsf{f}][t_1] = t_2$, and $hashFunction[\mathsf{g}][t_2] = t_3$. Finally the assignment to y sets $hashValue[\mathsf{y}] = t_3$.
2. We already have $hashValue[\mathsf{y}] = t_3$, so assign $hashFunction[\mathsf{g}][t_3] = t_4$. The assignment to z sets $hashValue[\mathsf{z}] = t_4$.
3. We have $hashValue[\mathsf{z}] = t_4$. So, we add t_4 to $nonNullExprs$. We create a new temporary variable $\mathsf{gvnTmp}^{\#}$, and construct an extra assignment $\mathsf{gvnTmp}^{\#} := \mathsf{z}$, and add it after the assume statement. Because $hashValue[\mathsf{z}] = t_4$, we also add $defaultVar[t_4] = \mathsf{gvnTmp}^{\#}$, which we will use later for substitutions to all expressions that hash to t_4.
4. We already have $hashValue[\mathsf{x}] = t_1$ and $hashFunction[\mathsf{f}][t_1] = t_2$, so we set $hashValue[\mathsf{a}] = t_2$.
5. Since $hashValue[\mathsf{a}] = t_2$, $hashFunction[\mathsf{g}][t_2] = t_3$ and $hashFunction[\mathsf{h}][t_3] = t_4$, the hash value of the expression a.g.h is t_4. We also have $defaultVar[t_4] = \mathsf{gvnTmp}^{\#}$. At this point, we observe t_4 being in $nonNullExprs$ and substitute the RHS a.g.h with $\mathsf{gvnTmp}^{\#}$. Finally, we set $hashValue[\mathsf{b}] = t_4$.
6. Because $hashValue[\mathsf{b}] = t_4$ and $defaultVar[t_4] = \mathsf{gvnTmp}^{\#}$ and $nonNullExprs$ contains t_4, we replace the expression b with $\mathsf{gvnTmp}^{\#}$.

The resulting code is shown below.

```
1  y := x.f.g;
2  z := y.h;
3  assume (z ≠ Null);
4  gvnTmp# := z;
5  a := x.f;
6  b := gvnTmp#;
7  assert (gvnTmp# ≠ Null);
8  c.g := d;
```

Clearly, we retain the invariant that #-tagged variables cannot be NULL, and it is now straightforward to prove the safety of the assertion. We also note that the expression substitution is performed in a conservative manner. It is aborted as soon as a subexpression is assigned to. For example, at line 8, we encounter an assignment to the field g, so we remove g from the dictionary $hashFunction$. This has the effect of g acting as a new field, and all terms referenced by this field will now be assigned new terms.

The above transformation, in general, is performed on the entire procedure, not just a basic block to fully exploit its potential. This occurs in two steps. First, loops are lifted and converted to procedures (with recursion), so that the control-flow of each resulting procedure is acyclic. Next, we perform a topological sort of the basic blocks of a procedure and analyze the blocks in this order. This ensures

```
L1:                                L1:
   assume (x ≠ Null);                 assume (x ≠ Null);
   gvnTmp#_1 := x;                    gvnTmp#_1 := x;
   goto L3;                           goto L3;
L2:                                L2:
   assume (x ≠ Null);                 assume (x ≠ Null);
   gvnTmp#_2 := x;                    gvnTmp#_2 := x;
   goto L3;                           goto L3;
L3:                                L3:
   assert (x ≠ Null);                 gvnTmp#_3 := x;
                                      assert (gvnTmp#_3 ≠ Null);
```

Fig. 5. A program snippet before GVN (left) and after GVN (right)

that by the time the algorithm visits a basic block, it has already processed all predecessors of the block.

When analyzing a block, the algorithm considers all its predecessors and takes the intersection of their *nonNullExprs* list and *hashValue* map. This is because an expression is non-null only if it is non-null in all its predecessors and, further, we can use a term for a variable only if it is associated with the same term in all its predecessors. Finally, an important aspect of the algorithm is to perform a sound substitution at the merge point of basic blocks.

Consider the code snippet on the left in Figure 5. In this example, although x is available as a non-null expression in L3, we cannot substitute x in the assertion by either $\mathsf{gvnTmp}^{\#}_1$ or $\mathsf{gvnTmp}^{\#}_2$ because neither preserves program semantics. Instead, we introduce a new variable $\mathsf{gvnTmp}^{\#}_3$ and add the assignment $\mathsf{gvnTmp}^{\#}_3 := $ x right before the assertion in L3 and use that for substituting x. This is achieved by the map *var2expr* in the main algorithm. It maps a #-tagged variable to the expression that it substitutes. In the above program, suppose we assign the term t to the non-null expression x. Hence, *nonNullExprs*[L1] and *nonNullExprs*[L2] both contain t. We also have *defaultVar*[L1][t] = $\mathsf{gvnTmp}^{\#}_1$ and *var2expr*[$\mathsf{gvnTmp}^{\#}_1$] = x. Since t is available from all predecessors of L3, we know that this term is non-null in L3. The question is finding the expression corresponding to this term t and introducing a new assignment for it. At this point, the map *var2expr* comes into play. We pick a predecessor of L3, say L1. We look for the default variable of t and find *defaultVar*[L1][t] = $\mathsf{gvnTmp}^{\#}_1$, we then search for *var2expr*[$\mathsf{gvnTmp}^{\#}_1$] = x. At this point, we find that the expression corresponding to term t is x, and we introduce a new assignment $\mathsf{gvnTmp}^{\#}_3 := $ x at the start of L3 and use this for substitution of x in L3. The next section describes the algorithm formally.

4 Algorithm

We present the pseudocode of our program transformation in this section, as Algorithms 2 and 3. The transformation takes a program as input and produces a semantically-equivalent program with new #-tagged variables that can never point to NULL. This involves adding assignments for these new variables,

and substituting existing expressions with these variables whenever we determine that the substitution will preserve semantics. A proof of correctness of our transformation can be found in our full version [7].

Algorithm 2 Algorithm to perform GVN

1: $nonNullExprs = \{\}$ ▷ block → non-null terms in block
2: $var2expr = \{\}$ ▷ #-tagged variable → expression
3: $defaultVar = \{\}$ ▷ block, term → variable for substitution
4: $hashValue = \{\}$ ▷ block, variable → term
5: $hashFunction = \{\}$ ▷ operator, terms → term
6: $currBlock$ ▷ current block
7: **function** DoGVN
8:     **for** $proc$ in $program$ **do**
9:         **for** $block$ in $proc.Blocks$ **do**
10:             **for** $stmt$ in $block.Stmts$ **do**
11:                 **if** $stmt$ is "assert expr $\neq$ NULL" or "assume expr $\neq$ NULL" **then**
12:                     gvnTmp$^{\#}$ $\leftarrow GetNewTaggedVar()$
13:                     $s \leftarrow$ "gvnTmp$^{\#}$:= expr"
14:                     $block.Stmts.Add(s)$
15:                     $var2expr[block][$gvnTmp$^{\#}] \leftarrow$ expr
16:     **for** $proc$ in $program$ **do**
17:         $sortedBlocks \leftarrow TopologicalSort(proc.Blocks)$
18:         **for** $block$ in $sortedBlocks$ **do**
19:             $nonNullExprs[block] \leftarrow \bigcap_{blk \in block.Preds} nonNullExprs[blk]$
20:             $hashValue[block] \leftarrow \bigcap_{blk \in block.Preds} hashValue[blk]$
21:             $currBlock \leftarrow block$
22:             **for** $term$ in $nonNullExprs[block]$ **do**
23:                 expr $\leftarrow var2expr[defaultVar[blk][term]]$ ▷ for some $blk \in Preds$
24:                 gvnTmp$^{\#}$ $\leftarrow GetNewSpecialVar()$
25:                 $var2expr[$gvnTmp$^{\#}] \leftarrow$ expr
26:                 $s \leftarrow$ "gvnTmp$^{\#}$:= expr"
27:                 $block.Stmts.AddFront(s)$
28:         **for** $stmt$ in $block.Stmts$ **do**
29:             $stmt \leftarrow ProcessStmt(stmt)$
30:             **if** $stmt$ is "gvnTmp$^{\#}$:= expr" **then**
31:                 $term \leftarrow ComputeHash($expr$)$
32:                 $nonNullExprs[block].Add(term)$
33:                 $defaultVar[block][term] \leftarrow$ gvnTmp$^{\#}$

At a high level, the idea is to use assume and assert statements to identify non-null expressions. We introduce fresh #-tagged variables and assign these non-null expressions to them. Then, in a second pass, we compute a *term* corresponding to each expression. These terms are assigned in a manner that if two expressions have the same term, then they are equivalent to each other. If we encounter an expression e with the same term as one of the non-null expressions e', we substitute e with the #-tagged variable corresponding to e'.

We start by describing the role of each data structure used in Algorithm 2.

Algorithm 3 Helper Functions for DoGVN

1: **function** PROCESSSTMT(*stmt*)
2:     **if** *stmt* is "assume expr" or "assert expr" **then**
3:         expr ← $GetExpr$(expr)
4:         **return** *stmt*
5:     **else if** *stmt* is "v := expr" **then**
6:         $hashValue[currBlock]$[v] ← $ComputeHash$(expr)
7:         expr ← $GetExpr$(expr)
8:         **return** *stmt*
9:     **else if** *stmt* is "v.f := expr" **then**
10:         expr ← $GetExpr$(expr)
11:         v ← $GetExpr$(v)
12:         $hashFunction.Remove$(f)
13:         **return** *stmt*
14: **function** GETEXPR(expr)
15:     **if** expr is v **then**
16:         *term* ← $ComputeHash$(v)
17:         **if** $nonNullExprs[currBlock]$ contains *term* **then**
18:             **return** $defaultVar[currBlock][term]$
19:         **return** v
20:     **if** expr is v.f **then**
21:         v ← $GetExpr$(v)
22:         **return** v.f
23: **function** COMPUTEHASH(expr)
24:     **if** expr is v **then**
25:         **if** $hashValue[currBlock]$ does not contain v **then**
26:             $hashValue[currBlock]$[v] ← $GetNewTerm$()
27:         **return** $hashValue[currBlock]$[v]
28:     **else if** expr is v.f **then**
29:         *term* ← $ComputeHash$(v)
30:         **if** $hashFunction$[f] does not contain *term* **then**
31:             $hashFunction$[f][*term*] ← $GetNewTerm$()
32:         **return** $hashFunction$[f][*term*]

- *nonNullExprs* stores the terms corresponding to non-null expressions of a particular block.
- *var2expr* maps a #-tagged variable to the expression it is assigned to in each block. This will be used to perform sound substitution at merge points of basic blocks, as discussed in the last example of Section 3.2.
- *defaultVar* maps the term of to an expression to the #-tagged variable that will be used for its substitution. Whenever we compute the term for an expression, if the term is present in *nonNullExprs*, we will use *defaultVar* to find the #-tagged variable that will be used for the substitution.
- *hashValue* maps variables to terms assigned to them in a particular block.
- *hashFunction* stores the mapping from a field and a term to a new term. It is used to store the term for expressions with fields.
- *currBlock* keeps track of the current block (used in helper functions).

We explain the algorithm step by step.

1. Lines 8 - 15 – In this first pass of the algorithm, we search for program statements of the form "assert expr $\neq$ NULL" or "assume expr $\neq$ NULL". This guarantees that expr cannot be NULL after this program location under all executions. Hence, we introduce a new variable gvnTmp$^{\#}$ and assign expr to it. This mapping is also added to $var2expr$.

2. Line 17 – Before the second pass, we perform a topological sort on the basic blocks according to the control-flow graph. This is necessary since the information of $nonNullExprs$ for the predecessors of a basic block is needed before analyzing it. Note that control-flow graphs of procedures in our language must be acyclic (we convert loops to recursion), thus a topological sorting always succeeds.

3. Lines 18 - 27 – We compute the set of expressions that are non-null in all predecessors. These expressions will also be non-null in the current block. We also need the term for each variable in the current block, which also comes from the intersection of terms from all predecessors. Finally, for all the non-null expressions, we add an assignment since these expressions may be available from different variables in different predecessors, as discussed in Section 3.2.

4. Lines 28 - 33 – Finally, we process each statement in the current block. This performs the substitution for each expression in the statement ($GetExpr$ function in Algorithm 3). $GetExpr$ computes the term for the expression ($ComputeHash$ function in Algorithm 3), and if the term is contained in $nonNullExprs$, the substitution is performed. Finally, if we encounter a store statement, "v.f := expr", we remove all mappings w.r.t. f in $hashFunction$. So, for the future statements (and future blocks in the topological order), new terms will be assigned to expressions related to field f.

Following Algorithm 2, we generate a semantically equivalent program, and as we show in our experiments, will have improved precision with regard to alias analysis. The main reason behind this improvement is that these #-tagged variables can never contain aS_0 in their points-to set, hence NULL cannot flow through these variables in the analysis.

5 Experimental Evaluation

We have implemented the algorithms presented in this paper for the Boogie language [19]. Boogie is an intermediate verification language. Several front-ends are available that compile source languages such as C/C++ [17, 23] and C# [2] to Boogie, making it a useful target for developing practical tools. (For C/C++, we make the standard assumption that pointer arithmetic does not change the allocation site of the pointer, and thus can be ignored for the alias analysis [29]; due to space constraints we do not describe these details in this paper.)

Our work fits into a broader verification effort. The *Angelic Verification* (AV) project[4] at Microsoft Research aims to design push-button technology for finding software defects. In an earlier effort, AV was targeted to find null-pointer bugs [6].

---

[4] https://www.microsoft.com/en-us/research/project/angelic-verification/

Bench	Stats		SSA only		SSA with GVN		
	KLOC	Asserts	Time(s)	Asserts	Time(s)	GVN	Asserts
Mod 1	3.2	1741	9.08	61	11.37	0.88	17
Mod 2	8.4	4035	11.34	233	17.62	1.13	45
Mod 3	6.5	4375	10.26	617	19.43	2.15	52
Mod 4	20.9	7523	24.04	543	33.99	2.43	123
Mod 5	30.9	11184	35.02	1881	59.84	7.11	232
Mod 6	37.8	12128	35.94	2675	70.71	11.13	452
Mod 7	37.2	6840	36.88	1396	53.24	3.44	127
Mod 8	43.8	12209	28.91	2854	62.27	5.38	475
Mod 9	56.6	19030	60.05	5444	106.61	12.40	508
Mod 10	76.5	39955	171.43	2887	839.58	475.08	372
Mod 11	23.5	6966	49.17	875	69.10	10.14	103
Mod 12	14.9	8359	24.57	820	59.13	13.41	210
Mod 13	22.1	11471	38.27	869	87.07	24.03	248
Mod 14	36.2	18026	48.56	2501	149.60	41.93	478
Mod 15	19.4	20555	55.07	586	269.35	134.06	131
Mod 16	54.0	16957	62.86	2821	127.67	30.46	342
Total	**491.9**	**201354**	**701.45**	**27063**	**2036.58**	**775.16**	**3915**

Table 1. Results showing the effect of SSA and GVN program transformations on the ability of alias analysis to prove safety of non-null assertions.

Programs from the Windows codebase, in C/C++, were compiled down to Boogie with assertions guarding every pointer access to check for null dereferences. These Boogie programs were fed to a verification pipeline that applied heavyweight SMT-solver technology to reason over all possible program behaviors. To optimize the verification time, an alias analysis is run at the Boogie level to remove assertions that can be proved safe by the analysis. As our results will show, this optimization is necessary. The alias analysis is based on Andersen's analysis, as was described in Figure 2. We follow the algorithm given in Sridharan et al.'s report [26] with the extra constraint that #-tagged variables cannot alias with NULL, i.e. they cannot contain the allocation site aS_0. We can optionally perform the program transformation of Section 4 before running the alias analysis. Our implementation is available open-source[5].

We evaluate the effect of our program transformation on the precision of alias analysis for checking safety of null-pointer assertions. The benchmarks are described in the first three columns of Table 1. We picked 16 different modules from the Windows codebase. The table lists an anonymized name for the module (**Bench**), the lines of code in thousands (**KLOC**) and the number of assertions (one per pointer dereference) in the code (**Asserts**). It is worth noting that the first ten modules are the same as ones used in the study with AV [6], while the rest were added later.

We ran our tool, using either SSA alone or SSA followed by our GVN transformation, followed by the alias analysis. We list the total time taken by the tool (**Time(s)**), including the time to run the transformation, and the number of assertions that were *not* proved safe (**Asserts**). In the case of GVN, we also

---

[5] At `https://github.com/boogie-org/corral`, project **AddOns\AliasAnalysis**

isolate and list the time taken by the GVN transformation itself (**GVN**). The experiments were run (sequentially, single-threaded) on a server class machine with an Intel(R) Xeon(R) processor (single core) executing at 2.4 GHz with 32 GB RAM.

It is clear from the table that GVN offers significant increase in precision. With only the use of SSA, the analysis was able to prove the safety of 86.56% of assertions, while with the GVN transformation, we can prune away 98.05% of assertions. This is approximately a 7X reduction in the number of assertions that remain. This pruning is surprising because the alias analysis is still context and flow insensitive. Our program transformation crucially exploits the fact that programmers tend to be defensive against null-pointer bugs, allowing the analysis to get away with a very coarse abstraction. In fact, this level of pruning means that investing in more sophisticated alias analyses (e.g., flow sensitive) would have very diminished returns.

The alias analysis itself scales quite well: it finishes on about half a million lines of code in approximately 700 seconds with just SSA (86.56% pruning) or 2000 seconds with GVN (98.05% pruning). We note that there is an increase in the running time when using GVN. This happens because the transformation introduces more variables, compared to just SSA. However, this increase in time is more than offset by the improvement presented to the AV toolchain. For example, with the GVN transformation, AV takes 11 hours to finish the first 10 modules [6], whereas with the SSA transformation alone it does not finish even in 24 hours. Furthermore, AV reports fewer bugs when using just SSA because the extra computational load translates to a loss in program coverage as timeouts are hit more frequently.

6   Related Work

Pointer analysis is a well-researched branch of static analysis. There are several techniques proposed that interplay between context, flow and field sensitivity. Our choice of using context-insensitive, flow-insensitive but field sensitive analysis is to pick a scalable starting point, after which we add precision at low cost. The distinguishing factor in our work is: (1) the ability to leverage information from assume and assert statements (or branch conditions) and (2) specializing for the purpose of checking non-null assertions (as opposed to general aliasing assertions). We very briefly list, in the rest of this section, some of the previous work in adding precision to alias analysis or making it more scalable.

Context Sensitivity. Sharir and Pnueli [25] introduced the concept of *call-strings* to add context-sensitivity to static analysis techniques. Call strings may grow extremely long and limit efficiency, so Lhoták and Hendren [21] used k-limiting approaches to limit the size of call strings. Whaley and Lam [28] instead use Binary Decision Diagrams (BDDs) to scale a context sensitive analysis.

Flow sensitivity. Hardekopf and Lin [11] present a staged flow-sensitive analysis where a less precise auxiliary pointer analysis computes def-use chains which is used to enable the sparsity of the primary flow-sensitive analysis. The technique

is quite scalable on large benchmarks but they abstract away the assume statements. De and D'Souza [8] compute a map from access paths to sets of abstract objects at each program statement. This enables them to perform strong updates at indirect assignments. The technique is shown to be scalable only for small benchmarks, moreover, they also abstract away all assume statements. Finally, Lerch et al. [20] introduce the access-path abstraction, where access paths rooted at the same base variable are represented by this base variable at control flow merge points. The technique is quite expensive even on small benchmarks (less than 25 KLOC) and do not deal with assume statements in any way.

Other techniques. Heintze and Tardieu [13] improved performance by using a demand-driven pointer analysis, computing sufficient information to only determine points-to set of query variables. Fink et al. [9] developed a staged verification system, where faster and naive techniques run as early stages to prune away assertions that are easier to prove, which then reduces the load on more precise but slow techniques that run later. Landi and Ryder [18] use conditional may alias information to over-approximate the points-to sets of each pointer. Context sensitivity is added using k-limiting approach, and a set of aliases is maintained for every statement within a procedure to achieve flow-sensitivity. Choi et al. [3] also follows [18] closely but uses sparse representations for the control flow graphs and use transfer functions instead of alias-generating rules. To the best of our knowledge, none of these techniques are able to leverage assume statements to improve precision.

7 Conclusion

This paper presents a program transformation that improves the efficiency of alias analysis with minor scalability overhead. The transformation is proved to be semantics preserving. Our evaluation demonstrates the merit of our approach on a practical end-to-end scenario of finding null-pointer dereferences in software.

References

1. Andersen, L.O.: Program Analysis and Specialization for the C Programming Language. Ph.D. thesis, DIKU, University of Copenhagen (May 1994)
2. Barnett, M., Qadeer, S.: BCT: A translator from MSIL to Boogie (2012), seventh Workshop on Bytecode Semantics, Verification, Analysis and Transformation
3. Choi, J.D., Burke, M., Carini, P.: Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In: Principles of Programming Languages. pp. 232–245 (1993)
4. Cocke, J.: Global common subexpression elimination. In: Proceedings of a Symposium on Compiler Optimization. pp. 20–24. ACM, New York, NY, USA (1970)
5. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Trans. Program. Lang. Syst. 13(4), 451–490 (Oct 1991)
6. Das, A., Lahiri, S.K., Lal, A., Li, Y.: Angelic verification: Precise verification modulo unknowns. In: Computer Aided Verification (CAV). pp. 324–342 (2015)
7. Das, A., Lal, A.: Precise null pointer analysis through global value numbering. CoRR abs/1702.05807 (2017), http://arxiv.org/abs/1702.05807

8. De, A., D'Souza, D.: Scalable flow-sensitive pointer analysis for java with strong updates. In: ECOOP. pp. 665–687. Springer (2012)

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

10. Gulwani, S., Necula, G.C.: Global value numbering using random interpretation. In: Principles of Programming Languages, POPL. pp. 342–352 (2004)

11. Hardekopf, B., Lin, C.: Flow-sensitive pointer analysis for millions of lines of code. In: Code Generation and Optimization (CGO). pp. 289–298 (2011)

12. Hasti, R., Horwitz, S.: Using static single assignment form to improve flow-insensitive pointer analysis. In: Programming Language Design and Implementation (PLDI). pp. 97–105 (1998)

13. Heintze, N., Tardieu, O.: Demand-driven pointer analysis. In: Programming Language Design and Implementation (PLDI). pp. 24–34 (2001)

14. Horwitz, S.: Precise flow-insensitive may-alias analysis is np-hard. ACM Trans. Program. Lang. Syst. 19(1), 1–6 (Jan 1997)

15. Jones, N.D., Muchnick, S.S.: A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In: Principles of Programming Languages (POPL). pp. 66–74 (1982)

16. Kildall, G.A.: A unified approach to global program optimization. In: Principles of Programming Languages. pp. 194–206 (1973)

17. Lal, A., Qadeer, S.: Powering the static driver verifier using corral. In: Foundations of Software Engineering. pp. 202–212 (2014)

18. Landi, W., Ryder, B.G.: A safe approximate algorithm for interprocedural pointer aliasing. SIGPLAN Not. 39(4), 473–489 (Apr 2004)

19. Leino, K.R.M.: This is boogie 2 (2008), https://github.com/boogie-org/boogie

20. Lerch, J., Spth, J., Bodden, E., Mezini, M.: Access-path abstraction: Scaling field-sensitive data-flow analysis with unbounded access paths (t). In: Automated Software Engineering (ASE). pp. 619–629 (2015)

21. Lhoták, O., Hendren, L.: Evaluating the benefits of context-sensitive points-to analysis using a bdd-based implementation. ACM Transactions on Software Engineering and Methodology (TOSEM) 18(1), 3 (2008)

22. Microsoft: Static driver verifier, http://msdn.microsoft.com/en-us/library/windows/hardware/ff552808(v=vs.85).aspx

23. Rakamarić, Z., Emmi, M.: SMACK: decoupling source language details from verifier implementations. In: Computer Aided Verification (CAV). pp. 106–113 (2014)

24. Ramalingam, G.: The undecidability of aliasing. ACM Trans. Program. Lang. Syst. 16(5), 1467–1471 (Sep 1994)

25. Sharir, M., Pnueli, A.: Two approaches to interprocedural data flow analysis, chap. 7, pp. 189–234. Prentice-Hall, Englewood Cliffs, NJ (1981)

26. Sridharan, M., Chandra, S., Dolby, J., Fink, S.J., Yahav, E.: Alias analysis for object-oriented programs. In: Aliasing in Object-Oriented Programming. Types, Analysis and Verification, pp. 196–232. Springer (2013)

27. Steensgaard, B.: Points-to analysis in almost linear time. In: Principles of Programming Languages (POPL). pp. 32–41. ACM, New York, NY, USA (1996)

28. Whaley, J., Lam, M.S.: An efficient inclusion-based points-to analysis for strictly-typed languages. In: Static Analysis Symposium. pp. 180–195 (2002)

29. Zheng, X., Rugina, R.: Demand-driven alias analysis for c. In: Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 197–208. POPL '08, ACM, New York, NY, USA (2008)