Bachelor Thesis

Max Berndt

Flow-Insensitive Points-To Analyses for Frama-C Based on Tarjan’s Disjoint-Sets

March 10, 2014

supervised by:
Prof. Dr. S. Schupp
Dipl.-Ing. S. Mattsen

Hamburg University of Technology (TUHH)
Technische Universität Hamburg-Harburg
Institute for Software Systems
21073 Hamburg
Declaration

I, Max Berndt, solemnly declare that I have written this bachelor thesis independently, and that I have not made use of any aid other than those acknowledged in this bachelor thesis. Neither this bachelor thesis, nor any other similar work, has been previously submitted to any examination board.

Hamburg, March 10, 2014

Max Berndt
## Contents

1. Introduction  

2. Algorithms  
   2.1. Dimensions of Points-To Analyses  
   2.2. Andersen’s Algorithm  
   2.3. Andersen’s Algorithm: Example  
   2.4. Steensgaard’s Algorithm  
   2.5. Steensgaard’s Algorithm: Example  
   2.6. Andersen vs. Steensgaard  

3. Disjoint Set Data Structure  
   3.1. Union by Rank Heuristic  
   3.2. Path Compression Heuristic  
   3.3. Implementation with Arrays  

4. Steensgaard with Disjoint Sets  
   4.1. Preparation  
   4.2. Base Constraints  
   4.3. Simple Constraints  
   4.4. Distinct Empty Sets  
   4.5. Retrieving the Points-to Sets  
   4.6. How are Complex Constraints Treated?  

5. Evaluation  
   5.1. Case Study  
   5.2. Testing  

6. Conclusion  

A. Appendix - OCaml Source Code  
   A.1. Union-find Data Structure  
      A.1.1. makeSet  
      A.1.2. findSet  
      A.1.3. union  
   A.2. Unit Tests  
      A.2.1. monotone  

v
1. Introduction

Many software analysis procedures require to know, which program variables may alias each other (point to the same location). They need this information before they can start their analysis. A compiler, e.g., can perform certain optimizations if it knows that two variables alias each other. Consider the C-function in Figure 1.1. Two integer pointers $a$ and $b$ are passed. The function $foo$ prints $*a$ twice with a manipulation of $*b$ in between. Is it necessary to read $*a$ again in line 4 or can we use a cached version generated in line 1?

```c
1 foo ( int * a, int * b ) { read(*a) 
2 printf("%d\n", *a); print(*a) 
3 *b = 2; write(*b) 
4 read(*a) 
5 printf("%d\n", *a); print(*a) 
6 }
```

Figure 1.1.: Is it necessary to reload $*a$?

Imagine we had supersets of locations $a$ and $b$ can possibly point to. If the intersection of these sets were empty, we could conclude that $b$ definitively never refers to a location $a$ can point to, and, hence, is never able to change $*a$. In this case, $a$ and $b$ do not alias each other and reloading $*a$ is not necessary. The generation of such supersets is carried out by a points-to analysis or pointer analysis. The supersets are called points-to sets (pts). Now that we have all terminology, let us summarize the statement: pts($a$) ∩ pts($b$) = $\emptyset$ ⇒ line 3 in Figure 1.1 never changes $*a$.

Computing points-to sets should be performed quickly and the results should be sound. Precision is an additional property to consider. For applications like data flow analyses, precision is more important. But also in the context of optimization, a high precision can reduce the number of intersections of points-to sets, and thus, reduce the number of false aliases. Computing exact points-to sets considering each path in the control flow graph, can become infeasible for large software suites with millions of code lines. Luckily, this is not necessary for applications like optimizations. Algorithms that disregard the order of the statements are flow-insensitive and compute less precise points-to sets. The algorithm from Lars Andersen, see Section 2.2, and Bjarne Steensgaard, see Section 2.4, are flow-insensitive and thus less precise. For applications like compiler optimizations which we motivated above, and for certain program analysis (see Frama-C\textsuperscript{1}), it is sufficient to compute supersets as long as they are sound.

\textsuperscript{1}Framework for Modular Analysis of C programs, frama-c.com
In this thesis, we compare Andersen’s and Steensgaard’s algorithm and contribute an implementation of Steensgaard’s algorithm in OCaml that uses Tarjan’s disjoint sets [6] as an underlying data structure. It is also known as union-find data structure and provides a fast set of operations needed for our algorithm.

Since the analysis framework Frama-C is written in OCaml as well, our implementation can be adjusted to serve as a plug-in for Frama-C.

In Chapter 4.4, we introduce distinct empty sets that help use implement Steensgaard’s algorithm.
2. Algorithms

In a points-to analysis, one tries to identify possible locations a variable can reference. All known algorithms differ in terms of precision and efficiency and are categorized in dimensions, which are explained below. We will later consider algorithms that implement flow-insensitive pointer analyses.

2.1. Dimensions of Points-To Analyses

- Flow-Sensitivity
- Context-Sensitivity
- Soundness
- Definiteness / may vs. must

Flow-sensitive points-to analyses consider the control flow of a program and compute points-to information for each point in the program. In difference, flow-insensitive analyses disregard the order of the statements and compute points-to information for variables. The results of flow-insensitive analyses are thus less precise but can be computed more efficiently.
Flow-insensitive analyses declare either equality constraints that interpret variable assignments bidirectionally, which means that pointer information flows in both directions, or use inclusion constraints. Steensgaard’s algorithm is an instance of equality based analysis which usually uses a union-find data structure, see Chapter 4 for more details on such a structure. Andersen’s algorithm uses inclusion constraints (is subset-based), which means that it treats assignments unidirectionally. The statement $a := b$, for example, induces that points-to information of the right hand variable $b$ becomes a subset of the points-to set of the left hand variable $a$.

Context-sensitive analyses consider the calling context of an assignment. That means, if a function manipulates a reference, the corresponding points-to information is returned to the calling entry point only. In contrast, a context-insensitive analysis will return updated points-to sets to all possible entry points of that function. It is obvious that context sensitivity can be achieved in a flow-sensitive analysis only, as it requires per-statement results, rather than a single one at the end.

A points-to analysis is defined to be sound, if it ensures that for all $a \notin \text{pts}(b)$ it holds that $b$ never points-to $a$. Each target location $a$ that can be referenced in any execution path of a program by a variable $b$, must be listed in the points-to set of $b$. The set can be bigger, but all possible references must be in it. Andersen’s and Steensgaard’s algorithm are both sound [1] [5].
A may points-to analysis finds aliases that may instantiate during runtime. This holds for some execution branches but not necessarily for all. A must analysis, in contrast, reveals aliases that occur for every possible execution of the program.

2.2. Andersen’s Algorithm

Andersen’s algorithm is flow-insensitive and therefore disregards the order of the program statements. It is the most exact flow-insensitive analysis. However, the achieved precision is paid for a time complexity of $O(n^3)$, where $n$ is the number of statements [1]. Andersen solves a constraint system in which pointer assignments are expressed as subset constraints. There are essentially four types of pointer assignments, which are listed together with their corresponding constraints in Figure 2.1. All other pointer variants can be modelled using temporal variables. A statement $a := **b$, for example, can be modelled with two statements $a := *t$ and $t := *b$.

<table>
<thead>
<tr>
<th>Constraint Type</th>
<th>Assignment</th>
<th>Constraint</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>$a := &amp;b$</td>
<td>$a \supseteq {b}$</td>
<td>loc(b) $\in$ pts(a)</td>
</tr>
<tr>
<td>Simple</td>
<td>$a := b$</td>
<td>$a \supseteq b$</td>
<td>pts(a) $\supseteq$ pts(b)</td>
</tr>
<tr>
<td>Complex</td>
<td>$a := *b$</td>
<td>$a \supseteq *b$</td>
<td>$\forall v \in$ pts(b) . pts(a) $\supseteq$ pts(v)</td>
</tr>
<tr>
<td>Complex</td>
<td>$*a := b$</td>
<td>$*a \supseteq b$</td>
<td>$\forall v \in$ pts(a) . pts(v) $\supseteq$ pts(b)</td>
</tr>
</tbody>
</table>

Figure 2.1.: Andersen’s constraints, assignments are treated unidirectional [2]

A very intuitive description of Andersen’s algorithm can be given, as a graph closure problem. In Algorithm 1, a graph drawing procedure is listed. This procedure will take a program and draw a graph. The final graph will contain sets that represent the points-to sets of each program variable in the program.

We initialize the graph using the Base and Simple constraints. Initialization starts by drawing a node for each program variable and an empty points-to set next to it. We then consider Base and Simple constraint. For a statement $a := &b$ we add $b$ to the points-to set next to node $a$ and for a statement $a := b$, we add pointees of $b$ to the points-to set next to node $a$.

We furthermore maintain a work set $W$ of nodes $v$ and apply two subroutines to them. These routines are listed in algorithm 1 and are verbosely explained in Section 2.3.
Algorithm 1: Andersen’s algorithm using a graph [2]

Data: constraints
Result: A collection of points-to sets

Initialize graph and points-to sets using base and simple constraints. See Section 2.3

Let $W = v | \text{pts}(v) \neq \emptyset$ (all nodes with non-empty points-to sets)

while $W$ not empty do
  $v \leftarrow$ select from $W$
  // Subroutine 1
  foreach $a \in \text{pts}(v)$ do
    foreach constraint $p \supseteq *v$ do
      add edge $a \rightarrow p$, and add $a$ to $W$ if edge is new
    end
    foreach constraint $*v \supseteq q$ do
      add edge $q \rightarrow a$, and add $q$ to $W$ if edge is new
    end
  end
  // Subroutine 2
  foreach edge $v \rightarrow q$ do
    $\text{pts}(q) = \text{pts}(q) \cup \text{pts}(v)$, and add $q$ to $W$ if $\text{pts}(q)$ changed
  end
end

2.3. Andersen’s Algorithm: Example

We explain Andersen’s algorithm by considering the following program as an example:

```
f := &g
a := &b
c := &d
a := c
e := *a
*a := f
d := &h
```

We start Anderson’s algorithm by initializing the graph using the Base constraints $a \supseteq \{c\}$, namely:

```
f := &g
a := &b
c := &d
d := &h
```
Each variable on the left hand side gets assigned the address of the variable on the right hand side. As a result, it points to that variable and, therefore, needs to have it in its points-to set. In Figure 2.2, node $a$ represents variable $a$. The corresponding points-to set is depicted just below and contains at this point variable $b$ only.

![Graph with Base constraints applied](image)

As a second step we continue by applying the Simple constraints. The program contains one statement that corresponds to a Simple constraint:

$$a := c$$

The corresponding constraint is $a \supseteq c$ and means that $a$ becomes a superset of $c$. Hence, in addition to its own elements, $a$ gets all elements that $c$ contains. Particularly, $d$ is contained in $c$ and therefore added to $a$ (Figure 2.3).

![Graph with Simple constraints applied](image)

We initialize our work set $W$ by adding all nodes, with a non-empty points-to set:

$$W := \{a, f, c, d\}$$

We take each element $v$ from $W$ and apply two different subroutines, (1) and (2):

- (1) For each element in the points-to set of $v$ handle constraints
- (2) Propagate points-to information along leaving edges of $v$
### Subroutine (1)

The first element from \( W \) is \( a \). Its points-to set is \( \{ b,d \} \). We start the subroutine with \( b \) by taking constraints that have \( *a \) on the right side. There is just a single one in this program:

\[
e := *a
\]

We draw an edge from \( b \) to \( e \) and add \( b \) to the work set \( W \):

\[
W := \{ a, f, c, d, b \}
\]

Now we search for constraints that have \( *a \) on the left side. Again, there is only one in this program:

\[
*a := f
\]

We draw an edge from \( f \) to \( b \) and add \( f \) to the work set \( W \). But since \( f \) is already contained in \( W \), nothing changes here.

Subroutine (1) is now applied to the second element in \( a \)'s points-to set, i.e. to \( d \), which results in the graph depicted in Figure 2.6. \( W \) stays unchanged.
Subroutine (2)

In Subroutine (2), edges that leave \( v \) are considered. But since \( a \) has no outgoing edge, nothing happens and we delete \( a \) from the work set \( W \) and start over with the next element:

\[
W := \{\emptyset, f, c, d, b\}
\]

The next element is \( f \) and we apply Subroutine (1) to it. There are no complex constraints in our program that have \( *f \) on the left or right hand side. In this example, Subroutine (1) will never change anything in any further run, since our sample program contains the two complex constrains treated above only. We therefore simply apply Subroutine (2) to all remaining elements in \( W \). However, in general Subroutine (1) is not effect-less at this point.

\( f \) has two outgoing edges, one towards \( b \) and one towards \( d \). We extend \( b \)'s and \( d \)'s points-to set by all elements that are in \( f \)'s points-to set. In other words, all elements from \( f \) are pushed along the arrows into \( b \)'s and \( d \)'s points-to set which is depicted in Figure 2.7.

Figure 2.6.: Graph after Subroutine (1) has finished

Figure 2.7.: Graph with g pushed from f to b and d
2.3. Andersen’s Algorithm: Example

Subsequently, we need to add $b$ and $d$ to $W$ if their points-to sets have changed. Indeed, they have changed but both are already in $W$, so $W$ is not adjusted:

$$W := \{ \emptyset, \emptyset, c, d, b \}$$

The next element $c$ has no leaving edge and is therefore deleted. We continue with all leaving edges of $d$ and "push" its points-to set contents to $e$’s point-to set (Figure 2.8).

Since $e$ is not contained in $W$, we add it now:

$$W := \{ \emptyset, \emptyset, \emptyset, \emptyset, b, e \}$$

We have one edge going from $b$ to $e$. Hence, we copy all elements from $b$’s points-to set into $e$’s and add $e$ to the work set (Figure 2.9).

Since $e$ has no leaving edge, nothing changes here. The work set $W$ is empty now and the algorithm finishes:

$$W := \{ \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset \}$$
As the result, the final points-to set of each variable is depicted next to the corresponding node in the graph. The points-to sets can be extracted from the graph and are summarized in Table 2.1.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Points-to set</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>{b, d}</td>
</tr>
<tr>
<td>b</td>
<td>{g}</td>
</tr>
<tr>
<td>c</td>
<td>{d}</td>
</tr>
<tr>
<td>d</td>
<td>{g, h}</td>
</tr>
<tr>
<td>e</td>
<td>{g, h}</td>
</tr>
<tr>
<td>f</td>
<td>{g}</td>
</tr>
<tr>
<td>g</td>
<td>{}</td>
</tr>
<tr>
<td>h</td>
<td>{}</td>
</tr>
</tbody>
</table>

Table 2.1.: Points-to sets computed by Andersen’s algorithm

### 2.4. Steensgaard’s Algorithm

Another flow-insensitive algorithm is the one from Steensgaard. Similar to Andersen’s algorithm, Steensgaard’s algorithm is also constraint-based but uses **equality constraints** instead of inclusion constraints. Therefore, it handles pointer assignments bidirectionally. Sets do not need to be updated, but can simply be merged. This leads to less precision but more efficiency. The algorithm can be executed in almost linear time [5] when a union-find data structure [6] is used.

Figure 2.10 lists the four assignment types and their corresponding Steensgaard constraints.
### 2.4. Steensgaard’s Algorithm

<table>
<thead>
<tr>
<th>Constraint Type</th>
<th>Assignment</th>
<th>Constraint</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>a := &amp;b</td>
<td>a ⊇ {b}</td>
<td>loc(b) ∈ pts(a)</td>
</tr>
<tr>
<td>Simple</td>
<td>a := b</td>
<td>a = b</td>
<td>pts(a) = pts(b)</td>
</tr>
<tr>
<td>Complex</td>
<td>a := *b</td>
<td>a = *b</td>
<td>∀ v ∈ pts(b) \ . pts(a) = pts(v)</td>
</tr>
<tr>
<td>Complex</td>
<td>*a := b</td>
<td>*a = b</td>
<td>∀ v ∈ pts(a) \ . pts(v) = pts(b)</td>
</tr>
</tbody>
</table>

Figure 2.10.: Steensgaard Constraints, pts(x) denotes the points-to set of variable x [2]

Steensgaard’s algorithm can, again, be seen as a graph closure problem. The graph drawing procedure is depicted in Algorithm 2. This procedure takes a program and draws a graph. The result is a directed graph with nodes as representatives for program variables. This time, points-to sets are represented as pointed-to nodes. For example $a \rightarrow b$ means that $\{b\}$ is the points-to set of $a$.

The graph is initialized with a node for each program variable. We then iteratively consider Basic, Simple, and Complex constraints. In each intermediate step, edges are drawn between nodes as stated in Algorithm 2. Steensgaard’s algorithm requires that each node has a fan-out $\leq 1$. This can lead to a recursive merging behaviour as sketched in Figure 2.11. Nodes that are pointees of a node $v$ with fan-out $> 1$ are merged. If, for example, $b$ and $e$ are merged, $c$ and $f$ become such pointees and thus, need to be merged as well.

Figure 2.11.: Merge of $b$ and $e$ leads to merge of $c$ and $f$
Algorithm 2: Steensgaard’s algorithm as a graph closure problem

Data: constraints
Result: points-to sets as pointees in the graph

Initialize graph with a node for each program variable

foreach basic constraint \( a \supseteq \{ b \} \) do
  add edge \( a \rightarrow b \)
  merge \( \text{pts}(a), b \) end

foreach simple constraint \( a = b \) do
  merge \( \text{pts}(a), \text{pts}(b) \) end

foreach complex constraint \( a = *b \) do
  \( v \in \text{pts}(b) \)
  merge \( \text{pts}(a), \text{pts}(v) \) end

foreach complex constraint \( *a = b \) do
  \( v \in \text{pts}(a) \)
  merge \( \text{pts}(v), \text{pts}(b) \) end

Procedure merge \( (setA, setB) \)
  if \( setA = setB \) then
    return
  else
    \( setA \cup setB \)
    merge \( \text{pts}(setA), \text{pts}(setB) \) end

2.5. Steensgaard’s Algorithm: Example

We consider again the same program as an example:

\begin{verbatim}
f := &g
a := &b
c := &d
a := c
e := *a
*a := f
d := &h
\end{verbatim}

We start Steensgaard’s algorithm by initializing the graph using the base constraints \( a \supseteq \{ c \} \), namely:
2.5. Steensgaard’s Algorithm: Example

\[ f := \&g \]
\[ a := \&b \]
\[ c := \&d \]
\[ d := \&h \]

Base constraints correspond to variable assignment as listed above. The variable left to
the equal sign extends its points-to set by the variable on the right side. In Figure 2.12,
e.g., node \( a \) represents variable \( a \). The corresponding points-to set is depicted as the
end of the arrow that leaves \( a \) and contains at this point variable \( b \) only. No node has
more than one outgoing edge, hence, we do not have to merge anything.

![Figure 2.12.: Graph with Base constraints](image)

In a second step, we continue by applying the Simple constraints, i.e.:

\[ a := c \]

The corresponding constraint is \( a = c \) and means that the two points-to sets of \( a \)
and \( c \) need to be unified (become equal). Particularly, \( d \) is contained in \( c \) and therefore
added to \( a \) (Figure 2.13).

![Figure 2.13.: Graph with Simple constraints](image)

Recall that all points-to relations must have a fan out \( \leq 1 \). This means, whenever a
node has more than one outgoing edge, the corresponding target nodes are merged into
a single one in order to fulfil the equality constraint. This is shown in Figure 2.14.
If this consolidation had lead to a new node with fan out > 1, the corresponding points-to nodes needed to be merged equivalently. Once all simple constraints have been covered, we continue by examining the complex constraints that have a * on either the left or right side:

\[
e := *a \\
* a := f
\]

The first one means, that \( e \) points just to all references, the content of \( a \) points to. Dereferencing \( a \) yields its content \{\( b, d \)\} which in turn points to \( h \). Hence, \( e \) points to \( h \) as well (Figure 2.15).

The second complex constraint \(*a = f\) specifies that the content of \( a \), i.e. \{\( b, d \)\}, points to the same location \( f \) points to. In this case, the location is \( g \) (Figure 2.16).
The node \( \{b,d\} \) has more than one outgoing edge. The targets \( g \) and \( h \) are merged to a single one, which is drawn in Figure 2.17.

As the result, the final points-to set of each variable is depicted as the target node of the corresponding node in the graph. A summary is given in Table 2.2.
2. Algorithms

<table>
<thead>
<tr>
<th>Variable</th>
<th>Points-to set by Andersen</th>
<th>Variable</th>
<th>Points-to set by Steensgaard</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>{b, d}</td>
<td>a</td>
<td>{b, d}</td>
</tr>
<tr>
<td>b</td>
<td>{g}</td>
<td>b</td>
<td>{g, h}</td>
</tr>
<tr>
<td>c</td>
<td>{d}</td>
<td>c</td>
<td>{b, d}</td>
</tr>
<tr>
<td>d</td>
<td>{g, h}</td>
<td>d</td>
<td>{g, h}</td>
</tr>
<tr>
<td>e</td>
<td>{g, h}</td>
<td>e</td>
<td>{g, h}</td>
</tr>
<tr>
<td>f</td>
<td>{g}</td>
<td>f</td>
<td>{g, h}</td>
</tr>
<tr>
<td>g</td>
<td>{}</td>
<td>g</td>
<td>{}</td>
</tr>
<tr>
<td>h</td>
<td>{}</td>
<td>h</td>
<td>{}</td>
</tr>
</tbody>
</table>

Table 2.2.: Points-to sets computed by Steensgaard’s algorithm

2.6. Andersen vs. Steensgaard

The two tables depicted in Table 2.3 illustrate the differences of the points-to sets computed by Andersen’s and Steensgaard’s algorithm. It is obvious that Steensgaard’s algorithm is less precise as it produces supersets of the points-to sets produced by Andersen’s algorithm.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Points-to set by Andersen</th>
<th>Variable</th>
<th>Points-to set by Steensgaard</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>{b, d}</td>
<td>a</td>
<td>{b, d}</td>
</tr>
<tr>
<td>b</td>
<td>{g}</td>
<td>b</td>
<td>{g, h}</td>
</tr>
<tr>
<td>c</td>
<td>{d}</td>
<td>c</td>
<td>{b, d}</td>
</tr>
<tr>
<td>d</td>
<td>{g, h}</td>
<td>d</td>
<td>{g, h}</td>
</tr>
<tr>
<td>e</td>
<td>{g, h}</td>
<td>e</td>
<td>{g, h}</td>
</tr>
<tr>
<td>f</td>
<td>{g}</td>
<td>f</td>
<td>{g, h}</td>
</tr>
<tr>
<td>g</td>
<td>{}</td>
<td>g</td>
<td>{}</td>
</tr>
<tr>
<td>h</td>
<td>{}</td>
<td>h</td>
<td>{}</td>
</tr>
</tbody>
</table>

Table 2.3.: Comparison of Andersen’s and Steensgaard’s result
However, for certain applications, like compiler optimizations, which were motivated in the introduction, less precise supersets may be sufficient. For example, \( \text{pts}(a) \cap \text{pts}(b) = \emptyset \) holds for both results calculated by Andersen’s and Steensgaard’s algorithm. Hence we are able use both results to conclude that:

\[ \text{pts}(a) \cap \text{pts}(b) = \emptyset \Rightarrow \text{line 3 in Figure 1.1 never changes} \]

Since computation of points-to sets using Steensgaard’s algorithm is remarkably faster, it should be considered for such an application. Chapter 3 introduces a data structure that helps us creating a high-performance implementation of Steensgaard’s algorithm.
3. Disjoint Set Data Structure

In this chapter we describe a disjoint set data structure that helps us implementing Steensgaard’s algorithm efficiently. Two sets are disjoint, if no set element occurs in both sets, or in other words, if their intersection is empty. A disjoint-set data structure maintains a collection of pairwise disjoint sets. This implies that an element will be in a single set of the collection only. Each set is identified by a representative, which is an arbitrary member of the set. The data structure supports the operations:

- **MAKE-SET**(x), which creates a new singleton set containing x only. x is also the representative of the set.
- **FIND-SET**(x), which returns a pointer to the representative of the set containing x. Since the sets are disjoint, x is contained in one set only. Therefore, the returned representative can be uniquely determined.
- **UNION**(x,y), which joins the two sets that contain x and y. A new representative is chosen.

We illustrate the disjoint set data structure in Figure 3.1. **MAKE-SET**(a), for example, creates a set depicted as a square containing just one element a, which is the representative of this set. **UNION**(c,d) unifies the two sets holding c and d and chooses a new representative for the resulting set. In this case c was picked, which is indicated by a bold font face. Lastly, the set operation **FIND-SET**(d) will return the representative of the set containing the argument d. In our example this is c.

![Figure 3.1.: Tarjan’s disjoint sets](image)

This data structure can be implemented using linked-lists, which results in a time complexity of O(m + n log(n)), where m is the total number of set operations and n the number of **MAKE-SET**(x) operations. However, if we use rooted trees instead of linked lists, we can achieve an almost linear time complexity [7]. Figure 3.2 depicts the sets as rooted trees. All connected nodes belong to a tree and are
thus elements of one set. \texttt{MAKE-SET}(a) creates a tree with depth 0 containing just the root element \(a\) at this point. The root is the representative of the corresponding set. The tree is illustrated as a graph, having nodes connected with arrows. Remember that \texttt{UNION}(c,d) merges the two sets containing \(c\) and \(d\). With trees, \texttt{UNION}(c,d) appends one tree to the root of the other. In fact, we use a union by rank heuristic to append the shorter tree to the longer to keep the depth of the resulting tree as low as possible. More details are explained in Section 3.1.

![Diagram of sets represented by rooted trees](image)

Figure 3.2.: Sets represented by rooted trees: The set \{b,c,d\} corresponds to the right tree

### 3.1. Union by Rank Heuristic

When unifying two trees, one could randomly append one tree to the root of the other. The problem with this strategy is that, if the deeper tree gets appended to the shorter, the resulting tree has a higher depth than the deeper tree had before. Executing many \texttt{UNION}(x,y) operations can clearly lead to trees that degenerate to a list. \texttt{FIND-SET}(x) operations on leaf elements would take more time to return the root, as they had to traverse all the way from the leaf element along the list toward the root. To prevent such a degeneration of trees, we collect rank information to utilize a \textbf{union by rank} heuristic. For each root, a maximum depth value is maintained. This value does not necessarily reflect the actual depth, but an upper bound. A \texttt{UNION}(x,y) operation appends the tree with the lower rank value to the root with the higher rank. In case both trees have the same depth, we pick one tree randomly, append it to the root of the other one and increment the rank value of the new root by one. The union by rank heuristic helps reducing the growth of trees during unification. An example is depicted in Figure 3.3. The tree with root \(a\) is appended to the deeper one with root \(b\).
3.2. Path Compression Heuristic

Keeping our trees as flat as possible, increases the performance of FIND-SET(x) queries. If a node \( v \) is a direct descendant of its root, \( \text{FIND-SET}(v) \) has time complexity of \( O(1) \), which is optimal. Therefore we try to maintain trees, whose leaves are direct children of the root element.

The path compression heuristic ensures that during \( \text{FIND-SET}(x) \) operations, each node on the way up to the root, is updated to point to the root directly. The effort updating each node on the path is worthwhile, since it linearizes the amortized runtime of later \( \text{FIND-SET}(x) \) operations. In other words, the access complexity of subsequent \( \text{FIND-SET}(x) \) queries will converge to \( O(1) \) due to this optimization. In Figure 3.4, the operation \( \text{FIND-SET}(e) \) traverses from node \( e \) to the root. All nodes detected on the path (node \( b \), \( d \) and \( e \)) are updated to point to the root directly. Rank information are not updated though, as the new depth is not known.
3. Disjoint Set Data Structure

3.3. Implementation with Arrays

We implement the disjoint sets using an array. This array is capable to store our disjoint-set forest by using integers as replacement for the set elements. Hence, if we for example want to store strings, we have to map them to integers and remap them back after the set operations. MAKE-SET(0..n) will initialize an array of size n + 1 as depicted in Figure 3.5. After initialization, the entry of each array field equals its index position. For example at index 0, a 0 is inserted. The index position represents a node in the forest and its entry denotes where this node points to. In the beginning, each node points to itself and is therefore a singleton set containing just itself.

![Figure 3.4: Path Compression Heuristic: Let all nodes on path from c to a point directly to a](image)

**Figure 3.4:** Path Compression Heuristic: Let all nodes on path from e to a point directly to a

**Figure 3.5:** Array and corresponding tree representation after MAKE-SET(0..5) is performed

![Figure 3.5: Array and corresponding tree representation after MAKE-SET(0..5) is performed](image)

Executing UNION(2,3) joins the two trees containing 2 and 3. In Figure 3.6 this is visualized by appending one tree to the root of the other (appending 3 to 2). In the array, we update the field at index 3 to point to 2 now. 2 still points to itself and is therefore identifiable as the root of the tree.

Arrays are very suitable for our disjoint sets, as they provide O(1) access times for any field in the array.
Figure 3.6.: Graph and array after Union(2,3) has been executed

Remember that \texttt{Find-Set}(x) returns the root of the tree that contains \(x\). In Figure 3.6, e.g., \texttt{Find-Set}(3) returns the root 2. This is implemented by reading the value of the index that equals the argument, i.e., the value of index 3, which is 2. \texttt{Find-Set}(x) is recursively repeated until the value equals the index. Is this case, the root of the tree is identified and \texttt{Find-Set}(x) terminates by returning the root. In the example above, this happens in the second iteration in which the value of index 2 is 2 as well.
4. Steensgaard with Disjoint Sets

In Chapter 2 we have discussed Steensgaard’s algorithm and explained why it can be useful for problems like compiler optimization. In Chapter 3 we introduced a high-performance data structure for disjoint sets that provides exactly those mechanisms we need for Steensgaard’s pointer analysis: unifying sets. We also showed how one can implement these sets as an array, which has the advantage of quick access times.

In this chapter, we bring those techniques together and demonstrate how one can use Tarjan’s sets to implement Steensgaard’s pointer analysis efficiently. We also show how one can solve a problem that occurs with initialized pointers.

4.1. Preparation

Consider the following program as an example:

\[
\begin{align*}
  f & := & g \\
  a & := & dl \\
  a & := & f \\
  g & := & h \\
  d & := & k
\end{align*}
\]

Initially, we convert all variable names to integers:

\[
\begin{align*}
  f & \rightarrow 0 & d & \rightarrow 3 \\
  a & \rightarrow 1 & h & \rightarrow 4 \\
  g & \rightarrow 2 & k & \rightarrow 5
\end{align*}
\]

The result is a translated version of our program:

\[
\begin{align*}
  0 & := & \&2 \\
  1 & := & \&3 \\
  1 & := & 0 \\
  2 & := & \&4 \\
  3 & := & \&5
\end{align*}
\]

For the later application of Steensgaard’s algorithm, we prepare two arrays. The first one (blue, left side of Figure 4.1) will be used to store membership information of variables. It is the array we introduced in Chapter 3 for storing Tarjan’s trees. Each variable will become a member of a node in the points-to graph. We will save points-to information in the second array (red, right side, light).
We use `MAKE-SET(0..5)` to create six single trees, all of which contain just one element (Figure 4.2). Technically, this operation is implemented by filling the blue array with natural numbers matching their corresponding index. The red array is initialized with -1, meaning that there are no pointing roots in the beginning. Values in this array represent relevant references only, if their index corresponds to a root element. If a root gets appended to another one, its previous pointer is still contained in the red array but no longer treated as relevant information.

Next we start Steensgaard’s algorithm by considering the base constraints $a \supseteq \{b\}$ originating from statement of the form $a = &b$. The points-to information is updated in the red array, which is depicted by red arrows between the trees in Figure 4.3. Note that an entry $x$ at index position $i$ in the red array specifies a points-to relation from $i$ to $x$ if $x$ is a root. That is, we insert, for example, the value 2 at index position 0 in the red array because of the constraint $0 = 2$.

\begin{align*}
0 &:= &2 \\
1 &:= &3 \\
2 &:= &4 \\
3 &:= &5 
\end{align*}
4.3. Simple Constraints

Whenever we update points-to information in the red array, we potentially need to unify two sets. This happens when the corresponding points-to set is already initialized, i.e., red_array[x] ≠ -1. If this is the case, and we encounter a new constraint x = &y, we perform UNION(red_array[x], y) and update the pointer relation accordingly: red_array(x) = FIND-SET(y).

4.3. Simple Constraints

We continue by considering assignments of the form a := b, namely 1 := 0. The corresponding Simple constraint is 1 = 0 and means that both points-to sets of 0 and 1 are merged into a single one (become equal). The points-to sets are identified using the red and the blue array: FIND-SET(red_array[0]) = 2 and FIND-SET(red_array[1]) = 3 return the two set representatives. Subsequently, we execute UNION(2,3) on the blue array to join the two sets. That is, we choose a new representative (one of the former representatives for reasons of simplicity), which becomes the root of the new combined tree. The second tree not containing this representative is appended to the root of the first tree. In fact, we actually use a rank heuristic to determine which tree to append to the other. Details are covered in Section 3.1.
The unification of the two trees is performed in the blue array by updating index 3. To find the root of node 3, we perform FIND-SET(3) on this array. FIND-SET(3) will compute the root (representative) of 3 in the tree, by first looking at index position 3 and then following the pointers up to the root. Evaluating index 3 leads to index position 2, which in turn points to itself and, therefore, identifies itself as the root of the tree. Hence, FIND-SET(3) will return 2.

During execution of UNION(2,3) we have to remember that the former independent members 2 and 3 of the new set \{2,3\} had points-to relations that need to be merged as well. Particularly, 2 pointed to 4 and 3 pointed to 5. Therefore, we run UNION(4,5) after UNION(2,3). Since at least one of 4 and 5 has an empty points-to set (stored as -1 in the red array), we do not have to perform any further UNION(x,y) operation. However, in practice it can happen that UNION(x,y) operations trigger a chain of more unifications. As visualized in Figure 4.5, UNION(4,5) again appends one tree to the root of the other. This is implemented by updating index 5 in the blue array to reference index 4.

![Graph after UNION(4,5) has been executed](image)

Pointers of non-root nodes like the one from 3 to 5 do not represent relevant information anymore. We have merged nodes into sets and only sets point to other sets. In other words, only pointers having a root as their source serve as relevant knowledge for our algorithm. For performance reasons, we do not delete pointers though.

### 4.4. Distinct Empty Sets

The naive implementation seen so far has a problem. Consider the program listed in Figure 4.6. The correct points-to sets, a Steensgaard analysis would compute, are

\[
\text{pts}(0) = \text{pts}(1) = \text{pts}(2) = \{1\}.
\]
4.5. Retrieving the Points-to Sets

0 := &1
1 := 2
2 := 0

Figure 4.6.: Program where naive approach fails

So far we explained a simplified version of our implementation. The problem with this naive approach is that the order in which statements are analysed does matter. Recall that this should not be the case with a flow-insensitive analysis. Analyzing the program in Figure 4.6 with this naive approach from top to bottom results in an incorrect points-to set for variable 1:

\[
\begin{align*}
\text{pts}(1) &= \emptyset, \\
\text{pts}(0) &= \text{pts}(2) = \{1\}.
\end{align*}
\]

This imprecise result is returned due to the fact that we do not distinguish between empty points-to locations. When analyzing the second line, variable 1 and 2 do not point anywhere, but still refer to the same (empty) location. When the points-to location of variable 2 is updated in the third line, the location of variable 1 must be updated as well. This can be achieved if we introduce **distinct empty sets**. Figure 4.7 shows how we double the array size to initially declare a private empty set for each variable. The advantage is that we are now able to unify empty sets. Having this set up, a top to bottom analysis will unify the **empty** points-to sets of variable 1 and 2 in the second line of Figure 4.6. The last line will then correctly update their points-to locations to \{1\}. Complex constraint of the form \(a = *b\) or \(*c = d\) work as well, since \(a\) and \(d\) initially point to distinct empty locations.

![Figure 4.7: Make-Set(0,1,2): Distinct empty sets by doubling the array width](image)

4.5. Retrieving the Points-to Sets

Applications that might use our implementation are interested in the points-to sets. The sets are encoded in the two arrays. To retrieve the information, we create a third
4. Steensgaard with Disjoint Sets

(dark black) array. The extraction is done by running over all indices of the blue array and computing their root. The indices are then written to the black array at the index position that equals the root of the element. Computing the root for index 3 reveals 2, for example. Therefore, 3 is inserted at index 2 in the black array. Figure 4.8 depicts the final result in the black array.

Now that we have prepared the points-to sets in the black array, we apply the four steps to retrieve the points-to set of a variable \( x \) as listed in Algorithm 3.
4.6. How are Complex Constraints Treated?

Complex constraints are treated similarly to Simple constraints. They are of the form

\[ a := *b \]
\[ c* := d \]

In short, the side with the asterisk is dereferenced and then the constraint is treated like a Simple constraint.

For a detailed explanation, let us consider the example above. The first assignment
means that \( a \) points to all references the contents of \( b \) point to. Dereferencing \( b \) yields its contents \( x \), which in turn may point to a tree called \( s^2 \). The points-to sets of \( a \) (\( s^1 \)) and \( *b \) (\( s^2 \)) are merged into a single one (become equal). \( a \)'s points-to set can be retrieved from the arrays using \( \text{red\_array}[\text{FIND-SET}(a)] = s^1 \). For the other side, we first lookup \( *b \)'s target in the red array, which is \( x \). Secondly, \( x \)'s points-to set \( s^2 \) is again retrieved from the arrays calling \( \text{red\_array}[\text{FIND-SET}(x)] = s^2 \). In a last step, \( \text{UNION}(s^1, s^2) \) will modify the blue array in such a way that both sets are merged and share a single representative (root).

The second type of Complex constraints, i.e. \( c^* := d \), is handled in the same way. Here, the left side is dereferenced twice in the red array.
5. Evaluation

We evaluate our results in two steps. Firstly, we compare a Steensgaard analysis to one performed by Frama-C. Secondly, we describe which properties must hold for an implementation of Steensgaard’s algorithm and how they were tested.

5.1. Case Study

We compare the points-to sets Frama-C’s value analyser computes, to those that our implementation of Steensgaard’s algorithm computes. A common C procedure for swapping two values is used as a source program. The source code is listed in Figure 5.1. Frama-C’s value analyser can be triggered with the command `frama-c -val swap.c`, where `swap.c` contains the source code seen in Figure 5.1. The analyzer computes the following points-to sets: `pts(a) = \{c,e\}`, `pts(b) = \{d,f\}` and `pts(c) = \{2\}`. Variable `c` has a constant points-to location. For variable `e`, Frama-C computes an empty points-to set (`UNINITIALIZED`). Since Steensgaard’s algorithm merges points-to locations, it will merge the nodes `c` and `e`, as they are both assigned to variable `a`. In other words, the node representing variable `a` points to both nodes representing variable `c` and `e`. The fan-out of node `a` is bigger than one and hence, the two nodes are unified. The points-to set computed by Steensgaard’s algorithm of `e` is therefore not empty, but points wherever `c` points to. In this case it is \{2\}.

Each additional call of `swap` with different variables will lead to imprecise points-to sets of these variables. All of them will incorrectly contain 2 in addition to their actual points-to locations. We would like to avoid this imprecision but still have maximal performance.

To summarize, Frama-C’s analyser computes more precise points-to sets than Steensgaard’s algorithm. Since Steensgaard’s algorithm calculates supersets, the results are not incorrect but less precise. Nevertheless, call-by-reference emulation using top level pointers is a frequently used pattern in C. If, for example, a behavior for swapping two values is needed multiple times in a project, one usually creates a function to modularize this behaviour. The function can then be called from all points in the program where the functionality is required. Calling such methods is done using top level pointers. Since function calls are frequent in many programs, we would prefer if they would not decrease the precision of the points-to sets.

Manuvir Das [3] presented an adaptation of Steensgaard’s algorithm that avoids unifications at the top level of pointer assignments. This is helpful for programs with function calls, as they usually contain many top level pointer assignments. In Figure 5.1 for example, all pointer assignments are only on the first level. The approach of Das would, just like Frama-C’s implementation, return an empty points-to set for variable `e`. Similar to Andersen, Das uses inclusion constraints for top level pointer assignments. Equality constraints are used for pointer assignments of lower levels only, that is, for pointers that
dereference at least twice.
Marc Shapiro [4] proposed a categorization of variables. His algorithm merges nodes only if they belong to the same category. In our example program depicted in Figure 5.1, we could assign $c$ and $d$ to one category and $e$ and $f$ to another category. This way, we avoid the unification of the points-to nodes from $a$ and $b$. Hence, Shapiro’s idea could be applied to improve the precision here as well.

```c
#include<stdio.h>

void swap(int *a, int *b){
    /* Frama-C’s value analyser lists explicit assignments only. Therefore we create copies here. */
    int *copy_of_a = a;
    int *copy_of_b = b;
    int t = *a;
    *a = *b;
    *b = t;
}

int main(void){
    int c, d, e, f;
    swap(&c,&d);
    swap(&e,&f);
    c = 2;
    return 0;
}
```

Figure 5.1.: C-Program for value analysis

### 5.2. Testing

To verify that we implemented Steensgaard’s algorithm correctly, we determined properties that hold for all input programs. These properties are checked with analyses conducted on randomly generated programs. We use OUnit\(^1\) to test our implementation. OUnit allows the definition of unit tests that can be grouped to a test suite. Apart from some trivial implementation tests, the suite contains the following tests:

- Property: FIND-SET($x$) always returns a root element.
  Test: `findSet_rootEqualsIndex` verifies that FIND-SET($x$) indeed returns a root element, namely an element that points to itself. This is done for a random program.

\(^1\)OUnit is a unit test framework for OCaml, [http://ounit.forge.ocamlcore.org/](http://ounit.forge.ocamlcore.org/)
5.2. Testing

- Property: At any time of a program analysis, the depth of each tree must be less or equal to the rank. The rank is an upper bound of the depth.
  Test: rankUpperBound creates a random disjoint set forest using MAKE-SET(x), FIND-SET(x), and UNION(x,y). It computes the depth of each tree and then verifies for each tree that the rank of the root is indeed an upper bound. Specifically, that for all trees holds depth(tree) ≤ rank(root).

- Property: Using Steensgaard’s algorithm explained in Chapter 2, we hand-computed points-to sets of 9 sample programs, which can be found on the attached CD.
  Test: solveSteensgaard computes the points-to sets of the 9 sample programs and compares them to the manual computed results. It checks, for example, if the points-to sets of a program containing cyclic assignments, collapse to one single points-to set that points to itself.

- Property: Since Steensgaard’s analysis is flow-insensitive, the order of the statements analyzed does not matter. We assert that the points-to sets of a program equals the points-to set of its reversed version.
  Test: reversed creates up to 1000 random programs with a maximum of 100 variables and statements. For each program, it generates a reversed version and compares the points-to set of the original with the reversed one.

- Property: This test is for performance scores.
  Test: sizedTest can be used to measure the performance of our implementation. We generate a random program with 10 million statements and 10000 variables. Computing the points-to sets on a notebook\(^2\) took 18 seconds. For a program with 1 million statements and 1000 variables it returned the results in less than a second.

- Property: The number of variables in each points-to set increases monotonically with more program statements.
  Test: monotone generates up to 1000 random programs. For each program we compute the points-to sets. Afterwards, we consecutively add random statements to each program and assert that the cardinality of all points-to sets increases monotonically.

\(^2\)Intel Core i5, 8 GB Ram
6. Conclusion

In this thesis, we compare two known algorithms for flow-insensitive points-to analyses. We show that they differ in precision and complexity. Andersen’s algorithm offers a flow-insensitive analysis with a high precision. Steensgaard’s algorithm provides a high performance points-to analysis. The time complexity in the number of program statements becomes almost linear if one uses Tarjan’s disjoint sets as a data structure for Steensgaard’s algorithm. We implemented the disjoint sets with a single array. The usage of arrays with quick reading access, has the advantage of providing a high-performance data structure for the analyzes. The performance has been tested in Section 5.2. We showed that a functional programming language like OCaml is well suited for an implementation of Steensgaard’s algorithm. Since our program is written in OCaml, it could be used as a prototype for a Frama-C plugin. During development we encountered a problem that emerges from uninitialized pointers. Their referenced locations have to be unifiable regardless of whether these locations are empty or not. To overcome this problem, we introduced distinct empty sets that can be merged.
Bibliography


A. Appendix - OCaml Source Code

A.1. Union-find Data Structure

A.1.1. makeSet

type unionFind = {treeArr : int array; rankArr : int array}

(* val makeSets : int → unionFind = <fun> *)
let makeSets n = {treeArr = Array.init n (fun i -> i);
                  rankArr = Array.make n 0}

A.1.2. findSet

(* val findSet : int → unionFind → int = <fun> *)
let findSet e union_find =
  let treeArr = union_find.treeArr in
  let pointToRoot root = List.iter (fun i -> treeArr.(i) <- root) in
  let rec helper e l =
    let parent = treeArr.(e) in
    if e <> parent then helper parent (e::l) else begin
      (* base case: we hit the root node
         make all collected nodes on the path point to the root
         (path compression heuristic) and
         return the root afterwards *)
      pointToRoot parent l;
      parent
    end
  in
  helper e []
A.1.3. union

(* val union : int -> int -> unionFind -> unit = <fun> *)

let union x y union_find =
    let rootX = findSet x union_find in
    let rootY = findSet y union_find in
    let treeArr = union_find . treeArr in
    let rankArr = union_find . rankArr in
    let rankX = rankArr .( rootX) in
    let rankY = rankArr .( rootY) in
    (* rootY -> rootX *)
    if rankX > rankY then treeArr .(rootY) <- rootX
    else if rankX < rankY then treeArr .(rootX) <- rootY
    else begin
        (* rank heuristic
          same rank: append x to y and increase y’s rank *)
        treeArr .(rootX) <- rootY;
        rankArr .(rootY) <- (rankY+1)
    end
A.2. Unit Tests

A.2.1. monotone

let rec randomProgram statements vars = match statements with
| 0 -> []
| _ -> let a = Random.int vars in
      let b = Random.int vars in
      match Random.int 4 with
      | 0 -> Basic(a,b) :: randomProgram (statements-1) vars
      | 1 -> Simple(a,b) :: randomProgram (statements-1) vars
      | 2 -> ComplexR(a,b) :: randomProgram (statements-1) vars
      | _ -> ComplexL(a,b) :: randomProgram (statements-1) vars

let monotone () =
  let n = (Random.int 50) + 1 in
  let statements = Random.int 50 + 1 in
  let vars = Random.int 20 + 1 in
  let program = randomProgram statements vars in
  let pts = pointsToSets (steensgaard program) in
  let rec helper prg pts n = match n with
  | 0 -> ()
  | _ -> begin
    let increased = prg @ randomProgram 1 ((Array.length pts)/2 + 1) in
    let ptsIncreased = pointsToSets (steensgaard increased) in
    assert_equal (for_all_isSubsetOf pts ptsIncreased) true;
    helper increased ptsIncreased (n-1)
  end in
  helper program pts n

let test_monotone test_ctxt =
  let n = Random.int 1000 in
  for i = 0 to n do
    monotone ()
  done;;