Refining heap-shape information for Java programs using reachable types

by

David Kräutmann

Research Group Computer Science 2

RWTH Aachen University

First supervisor: Prof. Dr. Jürgen Giesl
Second supervisor: Prof. Dr. Thomas Noll

Advisor: Florian Frohn

This thesis is submitted on March, 2016 in partial fulfilment of the requirements for the degree of Bachelor of Science in Computer Science

Abstract

We introduce a *reachable type analysis*, a new path-sensitive, sound, interprocedual analysis that uses heap-shape information to overapproximate all possible types reachable from objects. The resulting reachable type set can then be used to further refine heap-shape information generated by other tools. A preliminary evaluation of the analysis by measuring the accuracy of dynamic dispatch resolution occurring in the analysis itself shows substantially improved results.

Contents

CHAPTER 1

INTRODUCTION

Shape analysis is an useful tool for many other program analyses such as compiler optimisation [1], parallelisation [2], or termination checking, where the termination of code that uses linked data structures can depend on said data structured being cycle-free. Existing methods such as in AProVE [3], Julia [4] or COSTA [5] currently do not perform a modular and path-sensitive type analysis.

We introduce a *reachable type analysis*, a sound interprocedual analysis which for each method generates a *method result* consisting of a *parametric type set* for the return value, each argument and all modified static fields, describing how the reachable types for all affected references change after an invocation of the method. This method result is parametric, meaning that it can then be instantiated with argument type information only when used, allowing for improved locality during analysis in addition to improved performance as each method has to be analysed only once. Furthermore it is possible to access the parametric type set of any local reference or static field at any code location.

Our analysis requires existing path-sensitive heap-shape information and can be used in further analyses, for example by reducing the set of possible concrete targets of a virtual method invoke or disproving the existence of a path from one reference to another. In particular it can then be invoked again on refined heap-shape information to get even more refined types.

In chapter 2, we give an overview over the abstract path representation used, our requirements for the heap-shape analysis and Soot [6], the framework used for our analysis. Chapter 3 describes the reachable types of a program at runtime, the structure of our static data-flow analysis in detail, proves its correctness and elaborates how reachable type information can be used to refine heap-shape information. Finally, chapter 4 compares the quality of dynamic dispatch resolution in the analysis itself with and without refinement. A short overview over related work is given in chapter 5 and areas of improvement are highlighted in chapter 6.

CHAPTER 2

Background

2.1 Partial functions

We require a notion of partial functions to model various operations that require lookup by a key.

Definition 2.1. A partial function $f : A \mapsto B$ is only defined on a subset $A' \subseteq A$. For all $x \in A$, either $f(x) \in B$ or $f(x)$ is undefined. Partial functions consist of 2-tuples $x \to y$ so that $f(x) = y$ and each x occurs at most once. We define the following operations on partial functions:

- The domain of definition of a partial function is given by $\mathrm{Dom}(f) = A'$
- The restriction $f|_S : A \cap S \mapsto B$ of $f : A \mapsto B$ to a set S is obtained by setting all elements in $A \setminus S$ to undefined
- The intersection $f \cap g : A_1 \cap A_2 \mapsto B$ of two partial functions $f : A_1 \mapsto B$ and $g : A_2 \mapsto B$ is given by $f \cap g = f|_{\mathrm{Dom}(g)}$
- The union $f \cup g : A_1 \cup A_2 \mapsto B$ is well-defined iff $\mathrm{Dom}(f) \cap \mathrm{Dom}(g) = \emptyset$. Then
$$(f \cup g)(x) = \begin{cases} f(x) & \text{if } x \in \mathrm{Dom}(f) \\ g(x) & \text{if } x \in \mathrm{Dom}(g) \end{cases} \square$$
- The set-minus operation $f \setminus g : A_1 \mapsto B$ is given by $f|_{A \setminus \mathrm{Dom}(g)}$
- The symmetric difference $f \oplus g : A_1 \cup A_2 \mapsto B$ is given by $(f \setminus g) \cup (g \setminus f)$

2.2 Paths

Representation of heap structure requires a way to model field and array access sequences. While in actual programs paths can be described by a set of finite sequences, static analysis requires a more permissive language that can model potentially infinite paths. First we introduce a few auxiliary definitions:

Definition 2.2. A *heap edge* $\mathcal{H} = \mathcal{F} \cup \{\text{arrayAccess}\}$ is either an instance field or a placeholder for an array index access, as in general we cannot statically determine the accessed array element.

Definition 2.3. A *heap sequence* $\mathcal{H}^*$ is a finite word over the alphabet containing all heap edges.

Definition 2.4. An (abstract) *heap path* $\mathcal{P}$ is a pattern over heap edges with the following syntax:

$$
\begin{array}{lcl}
\langle \textit{path} \rangle & ::= & \epsilon \\
 & \mid & \langle \textit{concretePrefix} \rangle\ \langle \textit{abstractSuffix} \rangle \\
\langle \textit{concretePrefix} \rangle & ::= & \epsilon \\
 & \mid & \langle \textit{heapEdge} \rangle.\langle \textit{concretePrefix} \rangle \\
\langle \textit{heapEdge} \rangle & ::= & \text{field} \mid \texttt{arrayAccess} \\
\langle \textit{abstractSuffix} \rangle & ::= & \epsilon \\
 & \mid & \{\ \langle \textit{fieldEntries} \rangle\ \},\ \langle \textit{abstractSuffix} \rangle \\
\langle \textit{fieldEntries} \rangle & ::= & \langle \textit{fieldEntry} \rangle \\
 & \mid & \langle \textit{fieldEntry} \rangle,\ \langle \textit{fieldEntries} \rangle \\
 & & \text{so that each } \langle \textit{heapEdge} \rangle \text{ occurs at most once in } \langle \textit{fieldEntries} \rangle \\
\langle \textit{fieldEntry} \rangle & ::= & \langle \textit{heapEdge} \rangle^{[n, \langle \textit{upperBound} \rangle]} \text{ where } n \in \mathbb{N} \\
\langle \textit{upperBound} \rangle & ::= & 0 \mid 1 \mid *
\end{array}
$$

Definition 2.5 (Semantics)**.** The semantics of the heap path syntax are defined by a set of inference rules and axioms. If a heap sequence $s \in \mathcal{H}^*$ matches a path $p \in \mathcal{P}$, then $s \Downarrow p$. Note that h^n is the n-fold repetition of h, i.e. $h^n = \underbrace{h \cdots h}_{n \text{ times}}$. In the following inference rules,

- h denotes a heap edge
- s and *pre* denote a heap sequence

- p denotes a heap path
- *suf* denotes an abstract suffix

Let adjustBounds be a function that reduces the lower and upper bounds of a matching heap edge by the appropriate amount

$$\text{adjustBounds}(hs, h^{[n,m]}, s) := hs \setminus \{h^{[n,m]}\} \cup \{h^{[\max(0,n-|s|_h),\max(0,m-|s|_h)]}\}$$

where for all $n : \mathbb{N}$, $* - n := *$ and $|s|_h$ denotes the amount of elements h in a sequence s.

$$\epsilon \Downarrow \epsilon \tag{P-EPSILON}$$

$$pre \Downarrow pre \tag{P-PREFIX-EMPTY}$$

$$\frac{s_2 \Downarrow suf}{pre.s_2 \Downarrow pre\ suf} \tag{P-PREFIX}$$

$$\epsilon \Downarrow h^{[0,n]} \tag{P-ENTRY-EPS}$$

$$\frac{n \leq 1}{h \Downarrow h^{[n,1]}} \tag{P-ENTRY-ONE}$$

$$\frac{m \geq n}{h^m \Downarrow h^{[n,*]}} \tag{P-ENTRY-MANY}$$

$$\frac{\exists h^{[n,m]} \in hs, s_1 \Downarrow h^{[n,m]} \qquad s_2 \Downarrow \text{adjustBounds}(hs, h^{[n,m]}, s_1), suf}{s_1.s_2 \Downarrow hs, suf}$$

(P-ENTRIES-INCOMP)

$$\frac{\exists h_1^{[n_1,m_1]} \in hs_1, s_1 \Downarrow h^{[n_1,m_1]} \qquad \forall hs_2 \in suf, \forall h_2^{[n_2,m_2]} \in hs_2, h_1 \neq h_2 \qquad s_2 \Downarrow suf}{s_1.s_2 \Downarrow hs_1, suf}$$

(P-ENTRIES-ORDER)

Definition 2.6 (Equivalence)**.** Two paths $p_1, p_2 : \mathcal{P}$ are equivalent (denoted $p_1 \equiv p_2$) iff for all heap sequences s, $s \Downarrow p_1$ iff $s \Downarrow p_2$.

Definition 2.7 (Concatenation)**.** The concatenation $p_1.p_2 : \mathcal{P}$ of two paths $p_1, p_2 : \mathcal{P}$ is an operation defined so that the following inference rule holds

$$\frac{s_1 \Downarrow p_1 \qquad s_2 \Downarrow p_2}{s_1.s_2 \Downarrow p_1.p_2} \tag{P-CONCAT}$$

Definition 2.8 (Alternation)**.** The alternation $p_1|p_2 : \mathcal{P}$ of two paths $p_1, p_2 : \mathcal{P}$ is an operation defined so that the following inference rules hold

$$\frac{s \Downarrow p_1}{s \Downarrow p_1|p_2} \quad \text{(P-ALTER-LEFT)}$$

$$\frac{s \Downarrow p_2}{s \Downarrow p_1|p_2} \quad \text{(P-ALTER-RIGHT)}$$

Remark 2.1*.* The actual implementation of heap paths has the following structure:

1. A concrete prefix sequence $pre \in \mathcal{H}^*$
2. An abstract suffix consisting of a 4-tuple with
   (a) a flag whether the suffix matches the empty path ϵ
   (b) a partial function $oa : \mathcal{H} \mapsto \{0, 1, *\}$ so that for every matching heap sequence, all heap edges h contained in the heap sequence occur at most $oa(h)$ times
   (c) a partial function $ua : \mathcal{H} \mapsto \mathbb{N}$ so that for every matching heap sequence, all heap edges h occur at least $ua(h)$ times
   (d) a partial order $f < g$ so that if $f < g$, then the sequence only matches if no f occurs after g

2.3 Heap-shape analysis

A shape analysis is a static analysis that determines information about allocated data structures in a program. We require that the used shape analysis is sound, i.e. that the determined information is true for every program input.

Definition 2.9. A *heap-shape analysis* is a path-sensitive analysis that provides a function reachableAt : $(Stmt, Refs) \to Refs \mapsto \mathcal{P}$ which given a statement and a base reference provides a partial function of all local variables referring to an object reachable from the base reference to a *heap path* describing where that object is reachable from the given base. *Path-sensitive* means that the analysis doesn't simply check whether x is reachable from y but returns an overapproximation of all paths to x from y.

As long as the used heap-shape analysis is path-sensitive and sound, it is irrelevant what analysis is used exactly. In principle even non-path-sensitive shape analyses can be used by assuming a path matching every heap sequence for all reference.

2.4 Soot

Soot [6] is a Java framework which provides a set of intermediate representations and APIs for analysis and optimisation of Java bytecode. The primary intermediate language used for analyses is Jimple; other representations include BAF, a stack-based language, Shimple, a static single assignment variant of Jimple, and Grimple, a form of Jimple suitable for decompiling.

Soot's execution is divided into packs, with each pack containing phases—phases being where the actual analyses are run. Usually execution follows the following order:

1. The Jimple body creation pack (`jb`) is applied to each method body, which transforms Java bytecode into Jimple code.
2. The 4 following whole-program packs are applied. These contain all interprocedual analyses as each phase in this pack runs exactly once.
   (a) call graph pack (`cg`)
   (b) whole-jimple transformation pack (`wjtp`)
   (c) whole-jimple optimisation pack (`wjop`)
   (d) whole-jimple annotation pack (`wjap`)
3. After that, a set of per-method jimple transformation, optimisation and annotation packs is applied (`jtp`, `jop` and `jap`)
4. Finally, the Jimple bodies are converted into the bytecode precursor BAF in the `bb` pack and tags are aggregated in the `tag` pack, for example making sure that for each line number, only one tag exists.

2.4.1 Jimple

Jimple [6] is a representation of Java bytecode which is well-suited for static analyses and program transformations. Firstly, Jimple is compact; it has essentially only 11 types of statements and 19 instructions in total, compared to more than 200 instructions in Java bytecode. Secondly, every statement is in 3-address form, i.e. instructions are kept as simple as possible, most of them having the form `x = y op z`. Additionally, the stack has been eliminated and replaced by local variables. All implicit references to stack positions are now explicit references to locals. Finally, all local variables are typed with either a primitive, class or interface type. Due to type erasure in Java bytecode, support for generic types is missing.

2.4.1.1 Language description

Jimple has 11 types of statements, out of which *assignStmt*, *identityStmt* and *invokeStmt* are relevant to this thesis.

- *assignStmt* is either an assignment of an *rvalue* to a local or an *immediate* to a static field, an instance field or an array reference. An *rvalue* is a static field or instance field access, an array reference, an *immediate* or an expression and an *immediate* is a local or a constant.
- *identityStmt* is used to assign locals to special values such as `this`, parameter references or a caught exception.
- *invokeStmt* invokes a method and ignores the result

An expression can be one of the following:

- an usage of a binary operator such as + on two immediates
- a type cast
- a check whether an immediate is an instance of some type
- one of the following method invocations
  - `specialinvoke`, which is, e.g., an invoke of a constructor method
  - `interfaceinvoke`, which is a call of a method defined in an interface
  - `virtualinvoke`, which is a regular method call
  - `staticinvoke`, which calls a static method
- an instantiation of a new reference type (note that this doesn't call the actual constructor)
- an instantiation of an array or multi-array of some type with dimensions stored in immediates.
- the length of an immediate in case the immediate is an array
- the negative of an immediate in case the immediate is a number

```
⟨stmt⟩           ::= ⟨assignStmt⟩ | ⟨identityStmt⟩ | ⟨gotoStmt⟩
                   |  ⟨ifStmt⟩ | ⟨invokeStmt⟩ | ⟨switchStmt⟩
                   |  ⟨monitorStmt⟩ | ⟨returnStmt⟩ | ⟨throwStmt⟩
                   |  ⟨breakpointStmt⟩ | ⟨nopStmt⟩

⟨assignStmt⟩     ::= local = ⟨rvalue⟩;
                   |  field = ⟨imm⟩;
                   |  local.field = ⟨imm⟩;
                   |  local[⟨imm⟩] = ⟨imm⟩;

⟨identityStmt⟩   ::= local := @this: type;
                   |  local := @parametern: type;
                   |  local := @exception;

⟨gotoStmt⟩       ::= goto label;

⟨ifStmt⟩         ::= if ⟨conditionExpr⟩ goto label;

⟨invokeStmt⟩     ::= invoke ⟨invokeExpr⟩;

⟨switchStmt⟩     ::= lookupswitch ⟨imm⟩ {
                      case value$_1$: goto label$_1$;
                      ...
                      case value$_n$: goto label$_n$;
                      default: goto defaultLabel; };
                   |  tableswitch ⟨imm⟩ {
                      case low: goto lowLabel;
                      ...
                      case high: goto highLabel;
                      default: goto defaultLabel; };

⟨monitorStmt⟩    ::= entermonitor ⟨imm⟩; | exitmonitor ⟨imm⟩;

⟨returnStmt⟩     ::= return ⟨imm⟩; | return;

⟨throwStmt⟩      ::= throw ⟨imm⟩;

⟨breakpointStmt⟩ ::= breakpoint;

⟨nopStmt⟩        ::= nop;
```

Figure 2.1: Jimple grammar—statements

```
⟨imm⟩           ::= local | constant

⟨conditionExpr⟩ ::= ⟨imm⟩ ⟨condop⟩ ⟨imm⟩

⟨condop⟩        ::= > | < | = | != | <= | >=

⟨rvalue⟩        ::= ⟨concreteRef⟩ | ⟨imm⟩ | ⟨expr⟩

⟨concreteRef⟩   ::= field | local.field | local[⟨imm⟩]

⟨invokeExpr⟩    ::= specialinvoke local.method(⟨imm⟩, ···, ⟨imm⟩)
                 |  interfaceinvoke local.method(⟨imm⟩, ···, ⟨imm⟩)
                 |  virtualinvoke local.method(⟨imm⟩, ···, ⟨imm⟩)
                 |  staticinvoke method(⟨imm⟩, ···, ⟨imm⟩)

⟨expr⟩          ::= ⟨imm⟩ ⟨binop⟩ ⟨imm⟩
                 |  (type) ⟨imm⟩
                 |  ⟨imm⟩ instanceof type
                 |  ⟨invokeExpr⟩
                 |  new refType
                 |  newarray(type)[⟨imm⟩]
                 |  newmultiarray(type)[⟨imm⟩] ··· [⟨imm⟩] []*
                 |  length ⟨imm⟩
                 |  neg ⟨imm⟩

⟨binop⟩         ::= + | - | * | / | % | rem | << | <<<
                 |  >> | & | | | cmp | cmpg | cmpl
                 |  ⟨condop⟩
```

Figure 2.2: Jimple grammar—expressions

CHAPTER 3

Reachable type analysis

3.1 Semantics

In order to write a static analysis for determining a reachable type set, we first need to define what *reachable types* are: the set of types actually reachable from some object at runtime.

Definition 3.1. The *reachable type set* $\mathcal{RTS} = \mathcal{JT} \mapsto \mathcal{P}$ is defined as a partial function from Java types $\mathcal{JT}$ to heap paths $\mathcal{P}$. In addition to the operations on partial functions (see definition 2.1) we define the following operations on reachable type sets:

- The sum of two reachable type sets $s_1, s_2 : \mathcal{RTS}$ is defined by

$$s_1 + s_2 := s_1 \oplus s_2 \cup \{ty \to s_1(ty) | s_2(ty) \mid ty \in \mathrm{Dom}(s_1 \cap s_2)\}$$

- Given a path $p : \mathcal{P}$ and a reachable type set $s : \mathcal{RTS}$ the type set with prepended path $p.s$ is defined by

$$p.s := \{t \to p.s(t) \mid t \in \mathrm{Dom}(s)\}$$

- Analogous the type set with appended path $s.p$ is given by

$$s.p := \{t \to s(t).p \mid t \in \mathrm{Dom}(s)\}$$

- The operation access : $(\mathcal{RTS}, \mathcal{H}^*) \to \mathcal{RTS}$ returns the reachable type set after accessing a sequence of heap edges and is defined by

$$\mathrm{access}(s, hs) := \{t \to p' \mid t \in \mathrm{Dom}(s), s(t) \equiv hs.p'\}$$

For the definition of runtime reachable types we require an exact representation of heap sequences; this is provided by *exact reachable type sets*.

Definition 3.2. An *exact reachable type set* $\mathcal{RTS}_{\text{exact}} = \mathcal{JT} \mapsto \mathscr{P}(\mathcal{H}^*)$ is a variant of reachable type sets used for theoretical considerations of runtime reachable types that replaces the overapproximating heap path $\mathcal{P}$ with a set of exact heap sequences $\mathscr{P}(\mathcal{H}^*)$.

- The sum of two exact reachable type sets $s_1, s_2 : \mathcal{RTS}_{\text{exact}}$ is defined by

$$s_1 + s_2 := s_1 \oplus s_2 \cup \{ty \to s_1(ty) \cup s_2(ty) \mid ty \in \text{Dom}(s_1 \cap s_2)\}$$

- Given a heap sequence $h : \mathcal{H}^*$ and an exact reachable type set $s : \mathcal{RTS}_{\text{exact}}$ the type set with prepended path $p.s$ is defined by

$$h.s := \{t \to \{h.h_s \mid h_s \in s(t)\} \mid t \in \text{Dom}(s)\}$$

- Analogous the type set with appended heap sequence $s.h$ is given by

$$s.h := \{t \to \{h_s.h \mid h_s \in s(t)\} \mid t \in \text{Dom}(s)\}$$

- The operation access : $(\mathcal{RTS}_{\text{exact}}, \mathcal{H}^*) \to \mathcal{RTS}_{\text{exact}}$ returns the exact reachable type set after accessing a sequence of heap edges and is defined by

$$\text{access}(s, h) := \{t \to hs \mid t \in \text{Dom}(s), hs = \{h' \mid h.h' \in s(t)\}, |hs| > 0\}$$

With $\mathcal{RTS}_{\text{exact}}$ we can then define the theoretical reachable types of a reference at runtime.

Definition 3.3 (Runtime reachable types)**.** Let $\mathcal{RT}_c : \mathit{Refs} \to \mathcal{RTS}_{\text{exact}}$ denote the exact reachable type set of some reference with the memory state c (i.e. register, stack, and heap content) and $\text{class}(r)$ be the most specific class of the object referenced by r. The fields of a class cls are denoted as $\mathcal{F}(cls)$ and the size of the array referenced by r_a is $|r_a|$. Then $\mathcal{RT}_c$ is defined as follows where r denotes a reference to a class and r_a an array reference:

$$\mathcal{RT}_c(r) := \{\text{class}(r) \to \{\epsilon\}\} + \sum_{f \in \mathcal{F}(\text{class}(r))} f.\mathcal{RT}_c(r.f)$$

$$\mathcal{RT}_c(r_a) := \sum_{i=0}^{|r_a|} i.\mathcal{RT}_c(r[i])$$

Remark 3.1. In case we encounter an cyclic reference such as in

```
List xs = new List();
xs.next = xs;
```

we have an infinite heap sequence set for `xs`

$$\{\texttt{List} \to \{\epsilon, \texttt{next}, \texttt{next.next}, \cdots\}\}$$

Remark 3.2. The runtime reachable types are exact. For all $r : Refs$ and heap sequences $path$ starting from r, the following holds:

$$\text{access}(RT_c(r), path) = RT_c(r.path)$$

3.2 Flow analysis

Our reachable type analysis is implemented as a Soot forward data flow analysis on a control-flow graph, returning a parametric method result that can be instantiated with varying argument sets. As the reachable types analysis requires analysis results of all invoked methods, each result is stored in a map of methods to analysis results. If a new method has to be analysed, the analysis is performed recursively and the result is added to the map for future use.

Definition 3.4. A *parametric type set* $\mathcal{PTS} = (\mathcal{RTS}, \mathbb{N} \mapsto \mathcal{P})$ is a 2-tuple of a reachable type set $\mathcal{RTS}$ as defined in definition 3.1 and a partial function $\mathbb{N} \mapsto \mathcal{P}$ from the natural numbers to heap paths used for storing placeholders for arguments where the heap path describes where the reference to the argument is reachable.

- The sum $pts_1 + pts_2 : \mathcal{PTS}$ of two parametric type sets $pts_1, pts_2 : \mathcal{PTS}$ is defined as

$$\begin{aligned}(rts_1, p_1) + (rts_2, p_2) := (&rts_1 + rts_2, p_1 \oplus p_2 \cup \\ &\{n \to p_1(n)|p_2(n) \mid n \in \text{Dom}(p_1 \cap p_2)\})\end{aligned}$$

- The function apply $: (\mathcal{PTS}, \mathcal{PTS}^*) \to \mathcal{PTS}$ is used to simulate the effects of a method invoke on the parametric type set. It is passed a sequence of parametric type sets for each passed argument and returns the updated parametric type set. The definition is

$$\text{apply}((rts, ps), args) := (rts, \emptyset) + \sum_{n \in \text{Dom}(ps)} ps(n).args_n$$

Example 3.1. The application of a parametric type set pts, given by

$$pts = (\{\texttt{List} \to \{\texttt{next}^{[0,*]}\}\}, \{0 \to \epsilon, 2 \to \texttt{value}\})$$

to a sequence of arguments *args* defined by

$$args = ((\emptyset, \emptyset), (\emptyset, \emptyset), (\{\texttt{Object} \to \epsilon\}, \emptyset))$$

is the following parametric type set

$$\text{apply}(pts, args) = (\{\texttt{List} \to \{\texttt{next}^{[0,*]}\}, \texttt{Object} \to \texttt{value}\})$$

Definition 3.5. A *parametric method result* $\mathcal{PMR} = (\mathcal{PTS}, \mathcal{PTS}^*)$ is a 2-tuple consisting of

1. a parametric type set containing the return value's reachable types
2. a sequence of parametric type sets with the updated arguments' reachable types

- The apply : $(\mathcal{PMR}, \mathcal{PTS}^*) \to \mathcal{PMR}$ operator for method results is defined by applying the argument sets on each underlying set:

$$\begin{aligned}\text{apply}((retPts, argsPts), args) :=&(\text{apply}(retPts, args),\\ &(\text{apply}(argsPts_i, args))_{i \in [0,|argsPts|]})\end{aligned}$$

- The sum of two parametric method results for the same method pmr_1, pmr_2 : $\mathcal{PMR}$ is the sum of its components:

$$(r_1, a_1) + (r_2, a_2) := (r_1 + r_2, (a_{1,i} + a_{2,i})_{i \in [0,|a_1|]})$$

Example 3.2. Consider the following Java code:

```java
public static List setValue(Object value, List xs) {
    xs.value = value;
    return xs;
}
```

Then the parametric method result for List setValue(Object,List) is

$$(\underbrace{(\emptyset, \{0 \to \texttt{value}, 1 \to \epsilon\})}_{\text{return value type set}}, (\underbrace{(\emptyset, \{0 \to \epsilon\})}_{\text{first argument}}, \underbrace{(\emptyset, \{0 \to \texttt{value}, 1 \to \epsilon\})}_{\text{second argument}}))$$

Definition 3.6. The *control-flow graph* $G(m) = (V \subseteq \mathit{Stmt}, E \subseteq V \times V)$ of a Jimple method m is a directed graph where the vertices are the method's Jimple statements and there exists an edge $v_1 \to v_2$ iff execution can progress from v_1 to v_2. Its only source is the first statement of the Jimple method and its sinks are the return points.

Example 3.3. To create a control-flow graph from Java code, it first has to be converted into Jimple statements.

```
public Object constructList(Object x) {
    List orig = new List();
    List xs = orig;

    for (int i = 0; i < 100; i++) {
        if (x != null) {
            xs.value = x;
        } else {
            xs.value = new Unit();
        }

        xs.next = new List();
        xs = xs.next;
    }
    return orig;
}
```

The loop structure and `if`/`else` branch are easily recognizable and statements are fairly complicated. For analysis puproses, the Java code has to be transformed into Jimple code. The following code is the Jimple equivalent of the Java code above. It can then be processed into a control-flow graph.

```
public java.lang.Object constructList(java.lang.Object) {
    ListUtils r0;
    java.lang.Object r1;
    List r2, $r3, $r5, r6;
    Unit $r4;
    int i0;

    r0 := @this: ListUtils;
    r1 := @parameter0: java.lang.Object;

    $r3 = new List;
    specialinvoke $r3.<List: void <init>()>();

    r2 = $r3;
    r6 = r2;
    i0 = 0;
 label1:
    if i0 >= 100 goto label4;
    if r1 == null goto label2;
    r6.<List: java.lang.Object value> = r1;
    goto label3;

 label2:
    $r4 = new Unit;
    specialinvoke $r4.<Unit: void <init>()>();
    r6.<List: java.lang.Object value> = $r4;

 label3:
    $r5 = new List;
    specialinvoke $r5.<List: void <init>()>();
    r6.<List: List next> = $r5;
    r6 = r6.<List: List next>;
    i0 = i0 + 1;
    goto label1;

 label4:
    return r2;
}
```

As we can see, our high-level Java code with a **for**-loop got transformed into unnested code reminiscent of Java bytecode and assembly. The **for**-loop became a flat construct:

```
label1:
    if i0 >= 100 goto label4;
    // loop body
    i0 =  i0 + 1;
    goto label1;
label4:
    return r2;
```

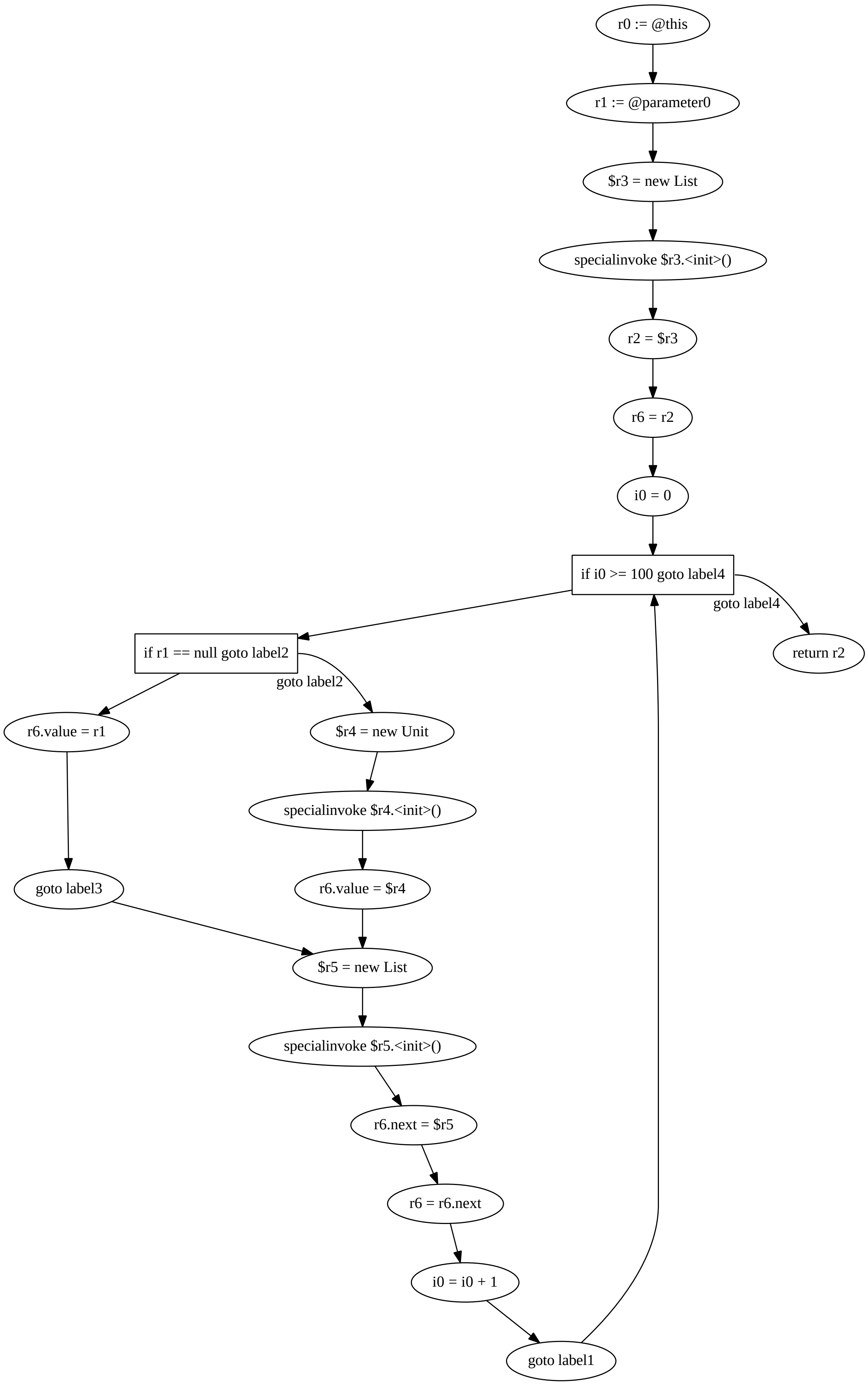
Similarly, the **if**/**else** statement also became a series of `goto`s:

```
if r1 == null goto label2;
// if body
goto label3;
label2:
// else body
label3:
// remainder
```

Note that there are some dissimilarities with Soot's Jimple output and our Jimple syntax. *External* Jimple output includes method and variable declarations which are not part of the actual Jimple statements and thus also not included in the control-flow graph.

```
public java.lang.Object constructList(java.lang.Object) {
    ListUtils r0;
    java.lang.Object r1;
    List r2, $r3, $r5, r6;
    Unit $r4;
    int i0;
    // ...
}
```

The resulting control-flow graph is given below:


r0 := @this
r1 := @parameter0
$r3 = new List
specialinvoke $r3.<init>()
r2 = $r3
r6 = r2
i0 = 0
if i0 >= 100 goto label4
goto label4
return r2
if r1 == null goto label2
goto label2
r6.value = r1
$r4 = new Unit
specialinvoke $r4.<init>()
goto label3
r6.value = $r4
$r5 = new List
specialinvoke $r5.<init>()
r6.next = $r5
r6 = r6.next
i0 = i0 + 1
goto label1

Here we can see the branches and loop structure again that became less obvious in the Jimple code, such as the loop structure in fig. 3.1 or the branch structure in fig. 3.2.

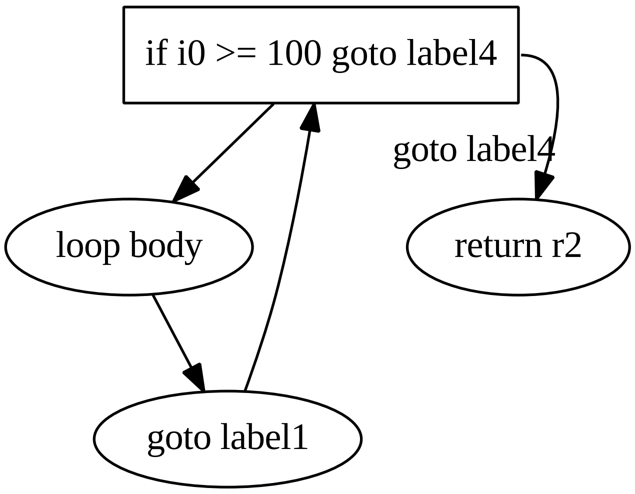


Figure 3.1: For loop in the control-flow graph

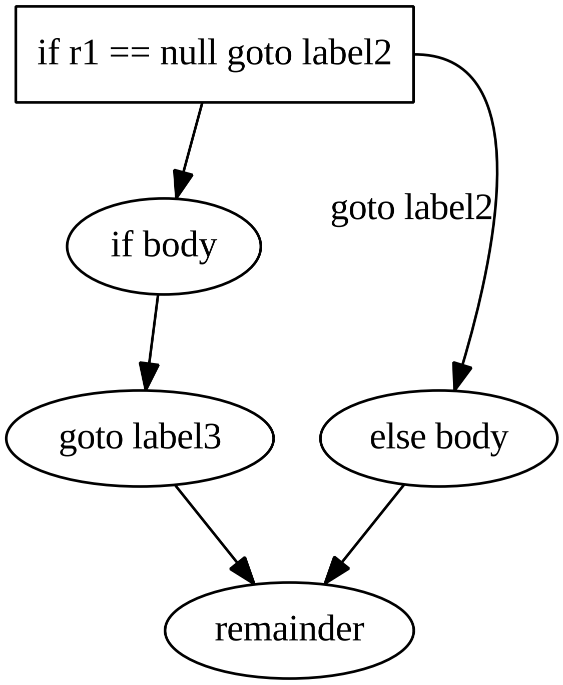


Figure 3.2: If/else branch in the control-flow graph

Definition 3.7 (Possible types of a right-hand value)**.** Given two partial functions $a, b : \mathit{Refs} \mapsto \mathcal{PTS}$ where a is the analysis state before the current statement and b is the new state and a Jimple *rvalue* (see section 2.4.1), the function $\mathrm{pt} : (\mathit{Refs} \to \mathcal{PTS})^2 \times \mathit{rvalue} \to \mathcal{PTS}$ is defined by performing case analysis on the *rvalue*:

Case 1: Local variables and static fields
For local variables and static fields, return the previous set in a. This set is always defined as uninitialised variables cannot occur.

Case 2: Instance fields and array accesses
For instance field accesses `r0.value` or array accesses `r0[idx]` retrieve the preceding type set pts for `r0` from a and return $\mathrm{access}(pts, \texttt{value})$ respectively $\mathrm{access}(pts, \texttt{arrayAccess})$. If there is no matching type set, then use the empty type set as the field is `null` if it never occurred in the base local's type set.

Case 3: *invokeExpr*
Invoke expressions are handled differently based on whether static or dynamic dispatch is used. For `specialinvoke` and `staticinvoke`, static dispatch is used; analysing the invoke target is sufficient.

However for dynamic invokes (`interfaceinvoke` and `virtualinvoke`), all possible concrete subclasses of the target method's class have to be analysed, with the results added together: If the type set of the base local contains a placeholder, analyse all possible implementations; otherwise it is sufficient to analyse the types whose paths match ϵ.

A reachable type analysis is performed on the invoked method (or a previously computed method result is retrieved), instantiating it with the invoke arguments' type sets. The result is then used to update the values and heap-shape predecessors of the passed arguments' sets like in the *assignStmt* case in algorithm 3.1, and the return set is used as the returned result.

Case 4: *castExpr*
The cast's underlying reference's set is returned.

Case 5:
For all other expressions such as `new`, `instanceof` and so on, the type of the expression is returned.

Example 3.4. We consider the possible types of a right-hand value after the first 3 statements in this method execute:

```java
public void example() {
    List xs = new List();
    xs.value = new Unit();
    Object obj = this;
}
```

The state after line 4 is then

$$a := \{\texttt{xs} \to (\{\texttt{List} \to \epsilon,\ \texttt{Unit} \to \texttt{value}\}, \emptyset), \texttt{thing} \to (\emptyset, \{0 \to \epsilon\})\}$$

and $b := a$. The possible types for different right-hand values after line 3 are then:

$$\begin{aligned}
\mathrm{pt}(a, b, r_0) &= (\{\texttt{List} \to \epsilon,\ \texttt{Unit} \to \texttt{value}\}, \emptyset) \\
\mathrm{pt}(a, b, r_0.\texttt{value}) &= (\{\texttt{Unit} \to \epsilon\}, \emptyset) \\
\mathrm{pt}(a, b, \texttt{setValue}(\texttt{obj}, \texttt{xs})) &= (\{\texttt{List} \to \epsilon,\ \texttt{Unit} \to \texttt{value}\}, \{0 \to \texttt{obj}\}) \\
\text{with } b &= \{\texttt{xs} \to (\{\texttt{List} \to \epsilon,\ \texttt{Unit} \to \texttt{value}\}, \{0 \to \texttt{value}\}) \\
&\quad , \texttt{obj} \to (\emptyset, \{0 \to \epsilon\})\} \\
\mathrm{pt}(a, b, \texttt{new Object}) &= (\{\texttt{Object} \to \epsilon\}, \emptyset)
\end{aligned}$$

where the parametric method result for `setValue` is computed in example 3.2.

The flow-through function is used to propagate the state along the control-flow graph.

Algorithm 3.1 (Flow-through function)**.** The function flowThrough : $(Refs \mapsto \mathcal{PTS}, Stmt) \to Refs \mapsto \mathcal{PTS}$ is defined for $a : Refs \mapsto \mathcal{PTS}$ and $stmt : Stmt$ as follows:

1. Initializing the new state with the preceding state $b := a$
2. Performing case analysis on the $stmt$:

   Case 1: ***assignStmt***
   If we have an *assignStmt*, which has the form `x = y`, use definition 3.7 to calculate the possible types $rvPts = \mathrm{pt}(a, b, y)$ of the *rvalue* `y`.

   Case 1.1: x is a local variable
   If `x` is a local variable, just set $b(\texttt{x}) = rvPts$

   Case 1.2: x is a field or array reference
   Otherwise `x` is a field or array reference. This case causes heap data to change, requiring us to take heap shape into account. Using the heap-shape analysis we calculate all predecessors of `x`, giving us a partial

function $preds : Refs \mapsto \mathcal{P}$ from references to paths. Then for each $v \in \mathrm{Dom}(preds)$, $b(v) = b(v) + preds(v).rvPts$

Case 2: *identityStmt*
For an *identityStmt* `local = @this` or `local = @parameter`n, set $b($`local`$) = (\emptyset, \{idx \to \epsilon\})$ where $idx = 0$ in the `@this` case, n in the `@parameter`n case when the method being analysed is static, or $n + 1$ otherwise.

Case 3: *invokeStmt*
An *invokeStmt* consists solely of an *invokeExpr*. This case is handled by invoking $\mathrm{pt}(a, b, invokeExpr)$.

3. Return b.

We require a framework in which we can apply our flow-through function. This is provided by the forward data flow analysis, which applies the flow-through function to every node of the control flow graph.

Algorithm 3.2 (Forward data flow analysis)**.** Given a control-flow graph $(V \subseteq Stmt, E)$ the *forward data flow analysis* creates two partial functions $a, b : Stmt \mapsto Refs \mapsto \mathcal{PTS}$ where a is the analysis state before the given statement and b is the state after. The analysis is performed by traversing the graph starting at the entry point. For each $stmt \in V$:

Case 1: $a(stmt)$ **undefined**
If $a(stmt)$ is undefined, we're at the entry point. Set $a(stmt) := \emptyset$ and go to case 2.

Case 2: $b(stmt)$ **undefined**
If $b(stmt)$ is undefined, the node hasn't been examined yet.
Set $b(stmt) = \mathrm{flowThrough}(a(stmt), stmt)$.

Case 3:
If both functions are defined at *stmt*, we perform a fixpoint iteration by setting $b(stmt) = flowThrough(a(stmt), stmt)$. If this did not cause a change in $b(stmt)$, we remove all predecessors of *stmt* from the control-flow graph to stop the fixpoint iteration.

After we set $b(stmt)$, we need to update the partial function a for all successors $\mathrm{succ}(stmt)$: For all $next \in \mathrm{succ}(stmt)$:

Case 1: $a(next)$ **undefined**
If $a(next)$ undefined, set $a(next) = b(stmt)$

Case 2:
Otherwise, set $a(next) = a(next) + b(stmt)$

We do not need to iterate manually, as loops in the control-flow graph already cause iterations.

Example 3.5. Using the code for `setValue`, we can now derive how the state after each line is computed. The `example()` method consists of the following Jimple statements:

```
r0 = @this;
r1 = new List;
specialinvoke r1.<init>();
r2 = new Unit;
specialinvoke r2.<init>();
r1.value = r2;
r3 = r0;
```

For simplicity, let s_n denote the statement at line n. The initial state $a(s_0)$ is empty. Applying the flow-through function to determine $b(s_0) = \text{flowThrough}(a(s_0), s_0)$ we can see that the *identityStmt* case is chosen. Thus $b(s_0) = \{\texttt{r0} \to (\emptyset, \{0 \to \epsilon\})\}$. Then we set $a(s_1) = b(s_0)$ and continue.

Calculating $b(s_1) = \text{flowThrough}(a(s_1), s_1)$ the *assignStmt* case in flowThrough is used. Then $\text{pt}(a(s_1), b(s_1), \texttt{new List}) = (\{\texttt{List} \to \epsilon\}, \emptyset)$ and $b(s_1) = b(s_0) + \{\texttt{r1} \to (\{\texttt{List} \to \epsilon\}, \emptyset)\}$

The constructors of `List` and `Unit` are empty, leading to $b(s_3) = a(s_3)$ and $b(s_5) = a(s_5)$. For s_4 the procedure is identical to s_2, leading to $b(s_4) = b(s_3) + \{\texttt{r2} \to (\{\texttt{Unit} \to \epsilon\}, \emptyset)\}$

Finally after s_7 we have state

$$\begin{aligned} b(s_7) = \{&\texttt{r0} \to (\emptyset, \{0 \to \epsilon\}), \texttt{r1} \mapsto (\{\texttt{List} \to \epsilon\}, \emptyset) \\ &, \texttt{r2} \mapsto (\{\texttt{Unit} \to \epsilon\}, \emptyset), \texttt{r3} \to (\emptyset, \{0 \to \epsilon\})\} \end{aligned}$$

We require a summary of the effects a method invocation has. This is given by the *parametric method result* from definition 3.5.

Definition 3.8 (Method result)**.** After performing the forward data flow analysis on a method with control-flow graph (V, E) with result $a, b : Stmt \mapsto Refs \mapsto \mathcal{PTS}$, a method result for the return points $s_r \in \text{sinks}(V, E)$ is a parametric method result consisting of:

1. The parametric type set at $b(s_r)$ for the returned reference, or the empty set if the function returns void

2. A sequence of sets at $b(s_r)$ for `this` and all arguments

The actual method result is then the sum of all individual method results.

Example 3.6. Given the following Java method with two return statements where in the first branch, only `String` is reachable and in the other branch the whole type set of argument x

```java
public Object removeNull(Object x) {
    if (x == null) {
        return new Nothing();
    } else {
        return x;
    }
}
```

the analysis result has to include both `Nothing` and the type set for `x`, as statically analysing which branch to choose is generally impossible here.

3.3 Correctness

Theorem 3.1. *The reachable type analysis is sound modulo static fields—all types that a reference can possibly reach during the execution of a program are computed by the flow analysis as long they do not involve static fields across method boundaries.*

Proof. Proof sketch Consider the control-flow graph (V, E) of a method m and let $a, b : Stmt \mapsto Refs \mapsto \mathcal{PTS}$ be the results of the forward data flow analysis on (V, E). For all statements $s \in V$ let $c(s)$ be the memory state directly after executing s and $\mathcal{RT}_{c(s)}$ the actual reachable types at the memory state $c(s)$ as in definition 3.3. Furthermore we require a sequence of method argument type sets $args : \mathcal{RTS}^*$ so that if m is a n-ary method (counting `this` as an argument), $s_0 := r_0 =$ `@this` or $s_i := r_i =$ `@parameter`i if m is static, $s_{i+1} := r_{i+1} =$ `@parameter`i otherwise, $|args| = n$ and $\forall i \leq n, \mathcal{RT}_{c(s_i)}(r_i) \subseteq args_i$. This ensures that we can instantiate all parametric type sets to a fully concrete reachable type set while maintaining soundness. Then for all references r in $\mathrm{Dom}(b(s))$ the proposition $\mathcal{RT}_{c(s)}(r) \subseteq rts$ with $(rts, \emptyset) = \mathrm{apply}(b(s)(r), args)$ has to hold in order for the analysis to be sound.

Note that $f \subseteq g$ for $f : \mathcal{RTS}_{\mathrm{exact}}$, $g : \mathcal{RTS}$ is defined as follows:

$$f \subseteq g := \forall ty \in \mathrm{Dom}(f), ty \in \mathrm{Dom}(g) \wedge \forall h \in f(ty), h \Downarrow g(ty)$$

Informally, the idea is that a set of heap sequences matches a heap path iff they all match—similar to path equivalence.

We perform induction on the construction of $\mathcal{RT}_{c(s)}(r)$

Case 1: Base: $\mathcal{RT}_{c(s)}(r) = \emptyset$
Trivial.

Case 2: Step: $\mathcal{RT}_{c(s)}(r) = R \cup \{\texttt{cls} \to hs\}$ **with** $R \subseteq rts$
As $R \subseteq rts$ by induction hypothesis, it remains to prove that $\{\texttt{cls} \to hs\} \subseteq rts$. First we consider the non-cyclic subset of hs. Then for all $h \in hs$ there exists a decomposition of the form $h = h_1. \cdots .h_n$ and a statement structure of the form

```
rhn = new cls;
...
rh1.h2 = rh2
ralias.field = rh1
r.h1 = ralias.field
```

or

```
rhi = @parameter //some identityStmt
...
rh1.h2 = rh2
ralias.field = rh1
r.h1 = ralias.field
```

where `ralias` is a placeholder for possible aliasing between each reference.

Case 2.1: All statements were intraprocedual

If all statements were an *assignStmt*, then the flow analysis follows each statement. The first statement's right-hand value $\text{pt}(a, b, \texttt{new cls})$ results in a parametric type set $(\{\texttt{cls} \to \epsilon\}, \emptyset)$ which is then propagated along the statements, with possible aliasing handled by the heap-shape analysis. As the heap-shape analysis is sound, we can conclude that $h \Downarrow b(s)(r)(cls)$

If one of the statements was an *identityStmt*, then by precondition on the argument type sets $args_i$, the possible types returned by $\text{pt}(a, b, r_i)$—where r_i is the local variable in the appropriate *identityStmt* as defined above—should be eventually replaced by the correct type set. However **due to an error when modularising the analysis, this does not work properly for r_i.`field`-type accesses.** In the non-parametric case for some reference x, the access x.`field` resolves to $(\emptyset, \emptyset)$ if no path matches `field`. This simulates the effect of an uninitialised field correctly on a non-parametric set, but not for parametric sets. A possible solution is given chapter 6—as this was only noticed during the final stages of the thesis there is insufficient time to provide a proper solution.

Case 2.2: Some statements occur in an invoked method

In case the invoke is statically dispatched, the correct method is chosen every time. Otherwise for a dynamic invoke $b.m(...)$ and assuming soundness for b then either all possible implementations out of the concrete reachable type set or all possible implementations period have been chosen, analysed and their method results added together.

The method result is then applied to the type sets of the arguments, replacing all eventual placeholders local to the scope of the invoked method with placeholders local to the scope of the invoking method. The resulting return and updated argument type sets then (barring the bug with argument field accesses) propagate the type changes. As every possible method has contributed to the method result, the actually chosen method's effects are also represented in the parametric type sets.

In case there exists a cyclic reference, *hs* is infinite. However it has to be periodic, i.e. *hs* is decomposable into a finite set of non-cyclic references hs_{fin} and a finite set of 2-tuples *cycles* consisting of a prefix h_{pre} and an infinite set hs_{suf} consisting of an iterated suffix h_{suf} so that

$$hs = hs_{\text{fin}} \cup \bigcup_{(h_{\text{pre}},hs_{\text{suf}})\in cycles} \; \bigcup_{h_{\text{suf}}\in hs_{\text{suf}}} h_{\text{pre}}.h_{\text{suf}}$$

Then the cycle is created by an assignment of the form $r_0.f = r_1$ where r_0 is a successor of r_1. By soundness of the heap-shape analysis this statement causes all paths involving r_0 and r_1 to be adjusted for the infinite cycles by setting the upper bound on all constituent fields to $*$.

We can conclude that $cls \in \text{Dom}(b(s))$ and $\forall h \in hs, h \Downarrow b(s)(cls)$.

□

Theorem 3.2. *The flow analysis tracks all execution paths that modify objects pointed to by references.*

Proof sketch. The intraprocedual case is trivial—this is handled by the control-flow graph. In case we encounter an *invokeExpr*, the static dispatch cases always call the same statically known function, and thus are not particularly interesting.

The dynamic dispatch case however takes exactly one code path at runtime but it is not always possible to decide what code paths it will take at compile time. If the method base object's type set contains any placeholders, it is not possible to determine exactly what subclass is called; thus, all subclass methods are analysed, including the actually used one. If the type set doesn't contain any placeholder then due to soundness of the reachable type analysis we also have the actually used subclass in the type set. □

Remark 3.3. The analysis is not complete — there can exist types in the type set that are unreachable during program execution; see example 3.6.

3.4 Refining heap-shape information

3.4.1 Call devirtualisation

Reachable type analysis can be used to resolve possible dynamic dispatch targets: Given a virtual invoke statement `r.s()` the set of possible concrete classes can be determined as follows: If the type set for r contains a placeholder, then every non-abstract subclass of the type of r has to be included. Otherwise it is sufficient to analyse all subclasses contained at ϵ. See example 3.7 for an example application in the reachable type analysis itself.

Example 3.7 (Resolving virtual invokes in the reachable type analysis). Consider a simplified class hierarchy implementing Jimple statements in Soot.

```
interface Stmt { void apply(Switch) }
abstract class DefinitionStmt implements Stmt { /* ... */ }
class IdentityStmt extends DefinitionStmt { /* ... */ }
class AssignStmt extends DefinitionStmt { /* ... */ }
class InvokeStmt implements Stmt { /* ... */ }
```

The possible types for `virtualinvoke x.apply(r)` where `x` has type `Stmt` and the type set for `x` is $(\emptyset, \{0 \to \epsilon\})$ are the union of the analysis results of all non-abstract implementations: `IdentityStmt.apply`, `AssignStmt.apply` and `InvokeStmt.apply`.

However, if the type set of `x` is $(\{$`IdentityStmt`$\to \epsilon\}, \emptyset)$ then the result is just the result of `IdentityStmt.apply`.

On the other hand if `x` has type `IdentityStmt` then the result is always given by the analysis of `IdentityStmt.apply`, and analogous with `AssignStmt`

3.4.2 Disproving path existence

Shape analysis frequently has to deal with the question whether there exists a path from the object referenced by x to the one referenced by y. This can be quickly disproven by checking whether the type of y is contained in the reachable type set of x. If it is not contained, then the analysis can immediately mark the path as non-existing due to contradiction; if there was a path, then it would have been picked up during the reachable type analysis.

CHAPTER 4

Results and evaluation

In order to evaluate the effectiveness of our analysis, we measure the improvement in virtual method resolution by analysing artificial problems[1] utilising dynamic dispatch. We then calculate the size all calculated parametric type sets and compare the average and maximum size with and without any refinement.

In the following table[2], let total_{ts}, avg_{ts} resp. max_{ts} be the total, average resp. maximum amount of types in the concrete type set and total_p, avg_p resp. max_p be the total, average resp. maximum amount of placeholders for method parameters. Furthermore *count* denotes the total amount of defined references in the analysis, i.e. using the definitions from algorithm 3.2 this corresponds to $\sum_{s \in \mathit{Stmt}} |\mathrm{Dom}(b(s,r))|$

As we can see, enabling refinement significantly reduced the amount of concrete types in the result while leaving the amount of abstract parameters unchanged. It also reduces the amount of references associated with a parametric type set, probably because some objects are only reachable if a specific method is called.

ResultListTest and Level1 use a local variable respectively a reference to a field of a local variable as the dynamic dispatch base reference. Thus the highest accuracy can be achieved here, as it's generally possible to uniquely restrict the called method since the heap paths should only consist of a finite sequence. In contrast to the first two test cases, FiniteListTest and LoopListTest iterate

[1]Originally it was planned to refine the used heap-shape analysis and use it to measure the quality of refinement. However due to implementation issues and time constraints integrating reachable type analysis with the heap-shape analysis was not possible at this point. Additionally, TPDB [7] problems did not utilize dynamic dispatch in a way that caused changes in the analysis results.

[2]The different test cases can be found at [8]

	Without refinement						
Test case	*count*	$total_{ts}$	$total_p$	avg_{ts}	avg_p	max_{ts}	max_p
ResultListTest	340	414	89	1.22	0.26	4	1
Level1	519	795	178	1.53	0.34	5	1
FiniteListTest	192	214	68	1.11	0.35	3	1
LoopListTest	343	547	77	1.59	0.22	3	1
	With refinement						
Test case	*count*	$total_{ts}$	$total_p$	avg_{ts}	avg_p	max_{ts}	max_p
ResultListTest	340	318	89	0.94	0.26	4	1
Level1	499	563	178	1.13	0.36	5	1
FiniteListTest	185	193	68	1.04	0.37	3	1
LoopListTest	336	526	77	1.57	0.23	3	1

Table 4.1: Test results

	Change in						
Test case	*count*	$total_{ts}$	$total_p$	avg_{ts}	avg_p	max_{ts}	max_p
ResultListTest	0%	-23%	0%	-23%	0%	0%	0%
Level1	-3.4%	-29%	0%	-26%	+4.0%	0%	0%
FiniteListTest	-3.6%	-9.8%	0%	-6.4%	+3.8%	0%	0%
LoopListTest	-2.0%	-3.8%	0%	-1.8%	+2.1%	0%	0%

Table 4.2: Result changes with refinement enabled

over a list when performing dynamic dispatch. FiniteListTest constructs the list sequentially while LoopListTest uses a loop. This causes the heap path of the base reference to be an abstract path, making dynamic dispatch devirtualization less exact.

Overall a reduction in parametric type set size was achieved every time, ranging from -3.8% to -29% in total size, signifying an increased analysis accuracy. Peculiarly the highest occurring type set size never changes, but that is likely an artifact of the used test cases.

CHAPTER 5

Related work

5.1 Heap-shape information

Separation logic [9] is an extension of Hoare logic for reasoning about shared data structures. Its key feature is *locality* where proofs or specifications of a subprogram in separation logic only mention the memory portion that is actually used in the subprogram. Applications include program analyses such as compositional shape analysis [10] which allows each procedure in a program to be analysed separately. The techniques developed in compositional shape analysis are in turn used e.g. in [11].

AProVE [3], Julia [4] or COSTA [5] all developed tools that deal with heap-shape information, whether using term rewriting systems in AProVE, abstract interpretation combined with constraint graphs in Julia or abstract interpretation with propositional formulae in COSTA.

Additionally, graph grammar methods use hyperedge replacement grammars [12] to obtain a finite hypergraph representation of data structures. This yields a finite state space that can then be further analysed with e.g. model checking tools.

5.2 Points-to analysis

Points-to analysis, also known as pointer analysis, is a static analysis that calculates the set of references that a reference can possibly take at runtime. Algorithms in weakly typed languages such as Steensgaard's method [13] cannot use type information as weakly typed languages support arbitrary casts. However in strongly typed, object-oriented languages such as C++ or Java algorithms such as rapid type analysis [14] or variable type analysis [15] do utilize and store type information, which can then be used for e.g. call devirtualization. These analyses either ignore fields completely in case of a *field-insensitive* points-to analysis or do not allow enumerating all fields at which some object or type occur as is the case for a *field-sensitive* points-to analysis.

CHAPTER 6

Future work

6.1 Current restrictions

Array references and static fields are currently not properly supported due to lack of implementation in the used heap-shape analysis. Array references are stored in the base reference whereas static fields are not propagated along method boundaries.

Recursion is currently unsupported due to both lack of implementation in the heap-shape analysis and the call graph traversal method used. This excludes a fairly large class of programs.

6.2 Known issues

Field access on parametric placeholders does not work as expected. This was unfortunately only noticed in the final stages of the thesis as the current behaviour is correct for non-parametric type sets. The following code currently returns $(\emptyset, \emptyset)$, which is not correct for parametric type sets.

```java
public static List getNodes(Tree tree) {
    return tree.nodes;
}
```

A possible solution would be to use parametric type sets $(\mathcal{RTS}, (\mathbb{N}, \mathcal{P}) \mapsto \mathcal{P})$ and extend the instance field case in pt (definition 3.7) so that the correct type set is returned. Parameter application would be defined by

$$\mathrm{apply}((rts, ps), args) := (rts, \emptyset) + \sum_{(n,pre)\in \mathrm{Dom}(ps)} ps(n).\mathrm{access}(args_n, pre)$$

Then the above code has a return type set of to $(\emptyset, \{(0, \texttt{nodes}) \to \epsilon\})$, which should work as expected. The soundness proof can be corrected by proving soundness of the access-operator, which should be possible as it is more restrictive on exact reachable type sets than parametric type sets.

6.3 Improvements

Type sets included as a dynamic dispatch possibility can be made conditional on the target type actually being in the callee's parametric type set when instantiating the parameters. Currently it only checks for possible targets at analysis and adds all targets if a set placeholder is encountered. The idea behind this is to emulate the more exact analysis results if it is re-run every time a method invocation is encountered, which predictably significantly worsens performance.

Another area of improvement is adding the reachable types of successors lazily; upon encountering an assignment of a reference to an instance field, reachable types of the right-hand side object are currently added to the reachable type set instantly. It could be prudent to add them to the reachable type set only when the original reference no longer exists or the method result is generated, as that allows the type set for e.g. `x.f = y; x.f = z;` to be safely reduced as `y` is no longer referenced.

6.4 Integration with heap-shape analysis

One of the primary goals when designing this analysis was to use it for refining a path-sensitive heap-shape analysis. This leads to a situation where the heap-shape analysis can use reachable type sets for e.g. call devirtualisation to generate a more exact heap model, while our reachable type analysis uses heap-shape analysis for information on predecessors of a reference. Iterating the two analyses can then lead to an increase in exactness at the cost of performance.

CHAPTER 7

Conclusion

We have developed a path-sensitive, sound, interprocedual reachable type analysis that can be used to improve existing tools and analyses. By utilising path-sensitive information from an existing heap-shape analysis in a data flow analysis we determine an overapproximation of all possible types reachable from a given reference along with paths describing where said types actually occur. We devised a data representation allowing for modular analysis with high data reuse based on an existing formulation of heap paths. Additionally we have proven the analysis sound and gave starting points on how to combine the reachable type analysis with other analyses. Evaluation based on dynamic dispatch resolution in the analysis itself shows a significant improvement in accuracy: up to 29% reduction in parametric type set size. Finally we provide key points on further improving the analysis, such as rectifying the inexactness in dynamic dispatch caused by the design choice to only run the analysis once on each method or integrating it with the used heap-shape analysis to allow for a tradeoff of performance with exactness.

Bibliography

[1] Y. Sui, Y. Li, and J. Xue. Query-directed adaptive heap cloning for optimizing compilers. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1–11, Feb 2013.

[2] Mark Marron, Mario Méndez-Lojo , Manuel Hermenegildo , Darko Stefanovic , and Deepak Kapur . Sharing analysis of arrays, collections, and recursive structures. pages 43–49. Association for Computing Machinery, November 2008.

[3] Marc Brockschmidt, Richard Musiol, Carsten Otto, and Juergen Giesl. Automated termination proofs for java bytecode with cyclic data. In *CAV*, January 2012.

[4] Enrico Scapin and Fausto Spoto. Field-sensitive unreachability and non-cyclicity analysis. *Sci. Comput. Program.*, 95:359–375, 2014.

[5] Samir Genaim and Damiano Zanardini. Reachability-based Acyclicity Analysis by Abstract Interpretation. *Theoretical Computer Science*, 474:60–79, 2013.

[6] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot-a java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.

[7] Termination problem data base. `http://cl2-informatik.uibk.ac.at/mercurial.cgi/TPDB`.

[8] Test cases for evaluation. `https://github.com/KaneTW/bachelor-thesis-testcases`.

[9] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society.

[10] Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. *SIGPLAN Not.*, 44(1):289–300, January 2009.

[11] Infer, a static analysis tool based on separation logic and bi-abduction. `http://fbinfer.com/docs/separation-logic-and-bi-abduction.html`.

[12] Jonathan Heinen, Christina Jansen, Joost-Pieter Katoen, and Thomas Noll. Verifying pointer programs using graph grammars. *Science of Computer Programming*, 97, Part 1:157 – 162, 2015. Special Issue on New Ideas and Emerging Results in Understanding Software.

[13] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 32–41, New York, NY, USA, 1996. ACM.

[14] David F. Bacon and Peter F. Sweeney. Fast static analysis of c++ virtual function calls. *SIGPLAN Not.*, 31(10):324–341, October 1996.

[15] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for java. *SIGPLAN Not.*, 35(10):264–280, October 2000.