DEMAND-DRIVEN STATIC ANALYSIS OF HEAP-MANIPULATING PROGRAMS

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Chenguang Sun

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

August 2019

Purdue University

West Lafayette, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF DISSERTATION APPROVAL

Prof. Samuel P. Midkiff, Chair

School of Electrical and Computer Engineering

Prof. Milind Kulkarni

School of Electrical and Computer Engineering

Prof. Vijay Raghunathan

School of Electrical and Computer Engineering

Prof. Charles A. Bouman

School of Electrical and Computer Engineering

Approved by:

Prof. Dimitrios Peroulis

Head of the School Graduate Program

To my loving and beloved parents, Chuanying Sun and Hua Chen.

ACKNOWLEDGMENTS

Acknowledgement is due to the people whose encouragement, advice, and support has been indispensable in the completion of this dissertation. My deepest gratitude goes to my advisor, Prof. Samuel Midkiff, for his guidance, support, and encouragement through my graduate study over the years.

Acknowledgement also goes to the committee members, Prof. Milind Kulkarni, Prof. Vijay Raghunathan, and Prof. Charles Bouman, for their generosity in sharing the expertise and taking the time to refine this work.

TABLE OF CONTENTS

	Page
LIST OF TABLES	vii
LIST OF FIGURES	viii
ABSTRACT	xii
1 INTRODUCTION	1
1.1 Large-Scale Code Base	2
1.2 Heap-Carried Data Flow	2
1.3 Implicit Control Flow Analysis	5
1.4 Asynchronous Control Flow Analysis	5
2 EXAMPLE LANGUAGE	11
3 EVOLUTION OF HEAP ANALYSIS TECHNIQUES	12
3.0.1 Global-Heap-Based Approach	12
3.0.2 The *CFL* Approach	15
3.0.3 Bottom-Up Summary-Based Approach	17
3.0.4 The FlowDroid Approach	19
4 CLIPPER – A NEW ON-DEMAND HEAP ANALYSIS	22
4.0.1 A Denotational Semantics for Procedure Alias Effect	22
4.0.2 A Specification based on Deduction Rules	25
4.0.3 A Loophole in FlowDroid	32
4.0.4 Static Fields and Jumping	35
4.1 Access Path Abstraction	40
4.1.1 Cyclic Pattern Reduction	42
4.1.2 Abstract Data Type Based Reduction	47
4.1.3 Soundness of Access Path Abstraction	50
5 IMPLICIT CONTROL FLOW ANALYSIS	51

Page

5.1 Limitation of EdgeMiner 53

6 DYNASENS – A DEMAND-DRIVEN POINTS-TO ANALYSIS 56

6.0.1 Overview of Parametric Points-to Analysis 56

6.0.2 Populating Input Relations 58

6.0.3 Experiment Setting 59

6.0.4 Downcast Safety Checking 62

6.0.5 Copy Constant Propagation 66

7 DYNASHAPE – A DEMAND-DRIVEN SHAPE ANALYSIS 69

7.1 The $\mathcal{LSL}$ Semantics 69

7.2 DynaShape 87

7.3 Soundness of Clipper as a Slicing Analysis 94

7.3.1 Clipper as a Deductive System for Alias Relations 96

7.3.2 $\mathcal{LSL}^{\mathcal{I}}$ – An Instrumented Small-Step $\mathcal{LSL}$ Semantics 99

8 ASYNCHRONOUS CONTROL FLOW ANALYSIS 123

8.1 ModShape 126

8.2 Asynchronous Control Flow Analysis 136

9 RELATED WORK 141

10 SUMMARY 142

REFERENCES 144

VITA 148

LIST OF TABLES

Table Page

3.1 Side-effect summaries of `Vector`'s methods. 18

6.1 Benchmark characteristics. The *Classes* and *Methods* columns show the number of classes and methods in the benchmark. The *Bytecodes* column shows the size of the benchmark in Kbytes. The *Queries* column shows the number of downcast sites in the application classes of the benchmark. . 63

6.2 Results for downcast safety checking. 65

6.3 Results for demand-driven copy constant propagation. The "Reachables", "Call-Graph", "Points-To" and "Time" columns have the same meaning as in Table 6.2. The "Safe Handlers" column shows the number of message handlers proven safe by the copy constant propagation analysis. 68

LIST OF FIGURES

Figure — Page

1.1 Example code illustrating downcast safety checking. 3

1.2 The heap model generated by the example program in Fig. 1.1. 4

1.3 Data flow generated by the example program in Fig. 1.1. 4

1.4 Example code illustrating implicit control flow. 6

1.5 Messaging framework of Android. 8

1.6 Message-handling in Android framework. 9

2.1 Syntax of a simple bytecode-like language. 11

3.1 Part of the data flows of a context-insensitive points-to analysis of the example program (Fig. 1.1). Numbers in boxes are the line number from Fig. 1.1. Part of the generated points-to graph is shown in the bottom-right corner. 13

3.2 An example *global* data flow (the red dotted arrow) between interfering load/store on Arr. 14

3.3 Example of realizable data flow from $i2$. The nodes represent local variable values[1]. Solid/dashed arrows represent forward/backward transitions between values incurred by the labeled statements. 16

3.4 Example code for recursive invocation. 17

3.5 Example illustrating FlowDroid. The solid/dashed arrows are explained in Chapter 4. 20

4.1 The base facts represented by the program elements. 26

4.2 Visualizing derivations as data flows. The derived facts $A(\alpha, X)$ are visualized as arrows labeled by X. Solid and dashed arrows are forward and backward data flows, respectively. 28

4.3 Visualizing derivations as data flows. The derived facts $A(\alpha, X)$ are visualized as arrows labeled by X. Solid and dashed arrows are forward and backward data flows, respectively. 28

4.4 Derivation examples. The numbers in parentheses are line numbers of corresponding statements. 33

Figure | Page

4.5 Example illustrating the loophole in FlowDroid. The forward data flows denoted by solid arrows are implemented by the taint analysis while the backward data flows denoted by dashed arrows are implemented by the on-demand alias analysis. The data flows missed by FlowDroid are marked in red. 34

4.6 Jumping example. The numbers in parentheses are line numbers of corresponding statements. 38

4.7 Visualizing jumping as data flows. The dotted red arrows represent jumping flows. 39

4.8 Example code for `List` interface. 40

4.9 Example code for illustrating access path abstraction. 41

4.10 Unbounded access paths generated by a loop iteratively. 42

4.11 Traversing recursive data structure via recursive method calls. 43

4.12 Unbounded access paths generated by recursive method calls. 44

4.13 Set of field paths encoded as a *DFA*. 45

4.14 String matching example. 45

4.15 Modified δ_f and $\delta_{\overline{f}}$ to support cycle reduction. 45

4.16 Bounded access paths abstracted via cycle reduction. 46

4.17 Multiple access paths caused by ADT. 48

4.18 Modified $\delta_f^{\#}$ and $\delta_{\overline{f}}^{\#}$ to support ADT-based reduction. 49

4.19 Examples of ADT-based field path reduction. 49

5.1 Inference of implicit control flows within the program of Fig. 1.4. The solid/dashed/red arrows have the same meaning as before. 52

5.2 Example code where a field-sensitive heap model is needed for implicit control flow analysis. 54

5.3 Example code where a field-sensitive heap model is needed for implicit control flow analysis. 55

6.1 Part of the data flows of the refined points-to analysis on the example program in Fig. 1.1. 60

6.2 Part of points-to graph generated by the refined points-to analysis. 61

7.1 The storeless heap model. 70

Figure Page

7.2 Example of store-based and storeless heap model. 71

7.3 Example code for illustrating intraprocedural $\mathcal{LSL}$. 71

7.4 An execution trace of the program in Fig. 7.3. The upper part of each storeless heap encodes the $Tran$ component and the lower part encodes the Gen component. For simplicity, the parameters of the $Tran$ component are omitted, i.e. $A = \{a1\} \rightarrow \{a1, b1.f\}$ is encoded as $A \rightarrow \{a1, b1.f\}$ . . . 72

7.5 Execution state and transition of the $\mathcal{LSL}$ semantics. 73

7.6 Intraprocedural $\mathcal{LSL}$ semantics. 75

7.7 Example of cycle created by heap store "$y.f = x$". 76

7.8 Example code for illustrating interprocedural $\mathcal{LSL}$. 77

7.9 An example trace of `main()`. 77

7.10 An example trace of `main()` (continued). Part of the heap generated in Fig. 7.9 is omitted. 78

7.11 An example trace of `foo1(A aa1)`. 79

7.12 An example trace of `bar(B b, int j1, int j2)`. 80

7.13 An example trace of `foo2(B bb2)`. 81

7.14 Another example trace of `bar(B b, int j1, int j2)`. 82

7.15 Interprocedural $\mathcal{LSL}$ semantics. 84

7.16 Example of cycle created by method call "$x = p(y1, y2)$". 88

7.17 Running CLIPPER with $i1$ as the slicing criteria. 90

7.18 Running CLIPPER with $i2$ as the slicing criteria. 91

7.19 DYNASHAPE analysis. 92

7.20 A tailored trace focusing on data flow to $i2$. 93

7.21 Part of the alias partition built from bottom up according to the program of Fig. 7.8. The arrows represent instantiation of callee's alias classes in caller's alias partition. 94

7.22 Part of the alias partition built from top down according to the program of Fig. 7.8. The arrows represent instantiation of caller's alias classes in callee's alias partition. 95

7.23 Basic facts and derivation rules for defining alias relation $\overline{R}$. 97

7.24 Execution state of the $\mathcal{LSL}^{\mathcal{I}}$ semantics. 100

Figure Page

7.25 $\mathcal{LSL}^{\mathcal{I}}$ semantics. 101

7.26 Definiton of same-level realizable path ($SLRP$). 110

7.27 Definition of realizable path (RP). 115

8.1 Example code for asynchronous messaging. 124

8.2 Asynchronous control flow graph of the example in Fig. 8.1. The nodes denotes message handling operations of the corresponding messages and the edges denotes message enqueuing operations representing the causal relation between the source and target messages. 125

8.3 Rewritten code of Android's messaging framework. 127

8.4 Header variable, packed object, and packed heap. 127

8.5 Packed heap of the message $\{target = Bar, what = 1\}$. 127

8.6 Shape rim and pack operation. 128

8.7 Example illustrating packing. 129

8.8 Pack transition of ModShape analysis. 130

8.9 Unpack transition of ModShape analysis. 131

8.10 Example illustrating unpacking. 131

8.11 Other intraprocedural transition of ModShape analysis. 132

8.12 Interprocedural transition of ModShape analysis. 133

8.13 Trace invariance and packed heap invariance. 134

8.14 Domains and helper functions of the ModShape analysis. 134

8.15 Equation system of the ModShape analysis. 135

8.16 Example illustrating asynchronous control flow analysis. The thick lines separate different traces. 137

8.17 Equation system of enqueuing summary. 139

8.18 Asynchronous control flow analysis. 140

ABSTRACT

Sun, Chenguang PhD, Purdue University, August 2019. Demand-Driven Static Analysis of Heap-Manipulating Programs. Major Professor: Samuel Midkiff.

Modern Java application frameworks present significant challenges for existing static analysis algorithms. Such challenges include large-scale code bases, heap-carried dependency, and asynchronous control flow caused by message passing.

Existing analysis algorithms are not suitable to deal with these challenges. One reason is that analyses are typically designed to operate homogeneously on the whole program. This leads to scalability problems when the analysis algorithms are used on applications built as plug-ins of large frameworks, since the framework code is analyzed together with the application code. Moreover, the asynchronous message passing of the actor model adopted by most modern frameworks leads to control flows which are not modeled by existing analyses.

This thesis presents several techniques for more powerful debugging and program understanding tools based on slicing. In general, slicing-based techniques aim to discover interesting properties of a large program by only reasoning about the relevant part of the program (typically a small amount of code) precisely, abstracting away the behavior of the rest of the program.

The key contribution of this thesis is a demand-driven framework to enable precise and scalable analyses on programs built on large frameworks. A slicing algorithm, which can handle heap-carried dependence, is used to identify the program elements relevant to an analysis query. We instantiated the framework to infer correlations between registration call sites and callback methods, and resolve asynchronous control flows caused by asynchronous message passing.

1. INTRODUCTION

People have been searching for methods to build robust software for decades [1]. The result is a corpus of theoretical and practical tools and methods. Since the advent of heap-manipulating programs, most programs' logic are tightly integrated with heap models. Hence these tools and methods are essentially designed as client analyses of underlying heap analyses for heap modeling and the efficiency of former relies on that of later.

The problem of heap modeling has been attacked from all angles with different kinds of heap analyses, from the long-standing points-to analyses [2] to the more recent shape analyses [3]. However, for all existing heap analyses, there exists certain trade offs between the precision of heap-modeling and the scalability of heap analysis. On the other hand, most existing client analyses and heap analyses in the literature have been proposed independently. Hence the trade-off adopted in most heap analyses may not be aligned with the demands from their client analyses.

Recently, increasingly more heap analyses are designed in a demand-driven style such that these heap analyses are customizable according to the demand from certain client analysis. However, most such customization methods are designed specific to certain heap analysis and cannot be generalized to and hence benefit other existing heap analysis methods.

Hence we propose a more general customization approach – applying a slicing analysis to identify all program elements relevant to given demand from certain client analysis and applying any existing heap analyses to the identified elements only.

In the proposed approach, a demand-driven heap analysis, called Clipper, is used as the slicing analysis and two demand-driven heap analyses – a points-to analysis called DynaSens and a shape analysis called DynaShape, both customized by

Clipper – are implemented to illustrate and evaluate the effect of the proposed approach.

The remainder of this chapter introduces two challenges in analyzing real-world programs – large-scale code bases (Section 1.1) and heap-carried data flow (Section 1.2) – as well as two features in many real-world frameworks to demonstrate the application of on-demand heap analysis – the callback mechanism (Section 1.3) and the message-driven mechanism (Section 1.4).

1.1 Large-Scale Code Base

Many popular framworks written in Java have a large code base consisting of a large number of classes and methods grouped in JAR archives and representing libraries. For example, an early version (2.3.7_r1) of the Android framework[1] alone consists of about 1.8M of bytecodes. Even seemingly simple applications can transitively depend on and thus trigger the loading of hundreds of classes because they transitively call methods defined in these libraries.

Such large code base presents significant challenges to existing static analysis algorithms, because existing analysis algorithms are typically designed to operate homogeneously on whole programs, starting from scratch at each analysis execution. For Java applications built with large libraries, the library code is analyzed together with the application code as part of the whole program. This creates potential scalability problems in terms of analysis time and memory usage, which limit the practical application of these analyses on real-world Java programs.

1.2 Heap-Carried Data Flow

Traditional data flow analyses only consider local variables [4]. However, local-only data dependence is very rare in programs written in modern programming languages

[1] https://www.android.com/

such as Java, which include heap load and store operations enabling data flow through the heap.

The example (Fig. 1.1), modified from the one in [5], is used to explain our points.

```
class Vector {
  Object[] arr;
  Vector() {
    Object[] a = new Object[10];
    this.arr = a;}
  Object get(int i) {
    Object[] a = this.arr;
    return a[i];}
  void set(int i, Object x) {
    Object[] a = this.arr;
    a[i] = x;
  }
}
class AddrBook {
  Vector names;
  AddrBook() {
    Vector v_names = new Vector();
    this.names = v_names;}
  void update(int i, String name) {
    Vector v_names = this.names;
    v_names.set(i, name);}
  String fetch(int i) {
    Vector v_names = this.names;
    return (String)v_names.get(i);}
}
void main() {
  Vector v_main = new Vector();
  Integer i1 = 3;
  v_main.set(0, i1);
  Integer i2 = (Integer)v_main.get(0);
  AddrBook book = new AddrBook();
  book.update(0, "bar");
  book.fetch(0);
}
```

Fig. 1.1.: Example code illustrating downcast safety checking.

,

The heap model generated by the example program in Fig. 1.1 is depicted in Fig. 1.2.

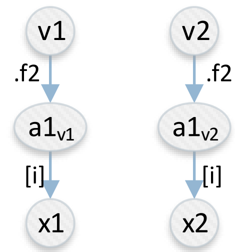


Fig. 1.2.: The heap model generated by the example program in Fig. 1.1.

Consider the flow of the integer 3 at line 28 to the cast site at line 30. Manual inspection of the source code shows the integer is first stored to the heap at line 11 and later loaded from the heap at line 8. The local data flow only propagates the `Vector` object `v1` directly, which references the integer indirectly via a sequence of field and array accesses, as illustrated in the data flow graph (Fig. 1.3).

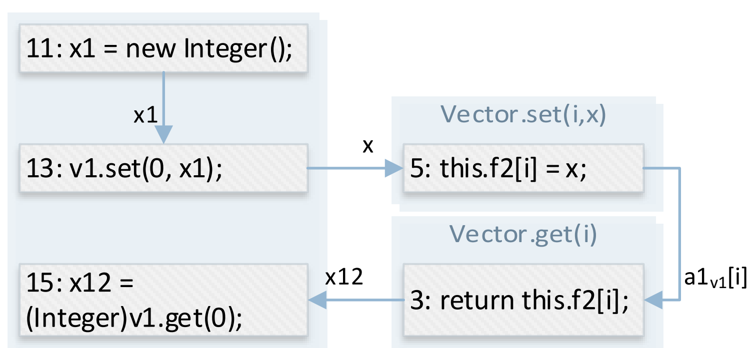


Fig. 1.3.: Data flow generated by the example program in Fig. 1.1.

Heap-carried data flows pose a challenge to the scalability and precision of static analyses. We present Clipper – an access-path based heap analysis – to resolve such heap-carried data flow on demand in Chapter 4.

1.3 Implicit Control Flow Analysis

Most frameworks provide registration interfaces for applications to register callback methods to interact with the framework. A callback method is implemented by, and hence part of, the application (e.g. `onPaused()` in Fig. 1.4b) but invoked by the framework (e.g. line 14 in Fig. 1.4a). Dually, a registration method is implemented by and hence part of the framework (e.g. `register()` in Fig. 1.4a) but invoked by the application (e.g. line 24 in Fig. 1.4b). In Fig. 1.4, for example, the `onPaused()` callback method is designed for receiving notification of the life-cycle event to pause the application that has registered the callback method with the `register()` method. Once the application is to be paused, the framework automatically invokes the registered `onPaused()` method.

One drawback for static analysis is that the callback methods do not have explicit incoming control flow within the application. Instead, control is transfered implicitly to the callback from the callback registration which notifies the framework of the existence of the callback method. On the other hand, an analysis considering the application's code only cannot recover such implicit control flows, i.e., analyses handling applications alone may generate incomplete control flow graphs with unreachable callback methods. The solution to such problems is explained in Chapter 5.

1.4 Asynchronous Control Flow Analysis

Concurrent programming is indispensable in distributed and multi-core environments. As one of the most popular computation model, the *Actor* model [6, 7] was designed specifically for programming in such environments. In this model, actors are essentially concurrent processes communicating with each other through asynchronous message passing.

In cooperative execution environments such as Erlang's runtime environment, actors are implemented as large number of concurrent processes that can be active

```
interface ICallback {
  void onPaused();
}
class App {
  List callbacks;
  void register(ICallback cb) {
    List list = this.callbacks;
    list.add(cb);
  }
  void dispatchPaused() {
    List list = this.callbacks;
    for (int i=0;i<list.size();++i) {
      ICallback cb = list.get(i);
      cb.onPaused();
    }
  }
}
```

(a) Framework

```
class MyCallback implements ICallback {
  void onPaused() {...}
}
class MyApp extends App {
  void onCreate() {
    ICallback mycb = new MyCallback();
    this.register(mycb);}
}
```

(b) Application

Fig. 1.4.: Example code illustrating implicit control flow.

simultaneously [8]. In non-cooperative execution environments such as Java Virtual Machine [9], instead of directly coupled to threads, actors are implemented in an message-driven style, with message handlers and messages representing actors and messages, respectively. In this case, one or more message dispatching threads running message loops can simulate all actors [10].

Unlike threads where causally related control flows are also textually related, in message-driven style, control flows are scattered into many cooperatively-triggered message handling functions, obscuring causal relation among them. This makes it hard to analyze and debug message-driven programs [10, 11].

The Android framework implements the *Actor* model in such message-driven style where a message is represented by a `Message` object. Fig. 1.5 shows the structure of the `Message` object and Fig. 1.6 shows two code snippets demonstrating enqueuing and processing of the `Message` object, respectively. The "`what`" field of a `Message` object (line 3) records an integer value denoting its *message type*. This field is written (line 12) before enqueuing the message and read (line 3) after dequeuing the message by a *handler*.

Fig. 1.5 shows the message handling framework of the Android system. Messages are represented with `Message` objects. The `target` field denotes the message handler and the `what` field denotes the message type. The message dispatching threads invoke the `loop()` method of the `Looper` class, which dequeues message objects from the message queue referenced via the `mQueue` field (line 15) and invokes `handle()` method on handlers of these message objects (line 17).

Fig. 1.6 shows a message passing example based on the message handling framework in Fig. 1.5. The `schedule()` method invokes the `send()` method to enqueue a message of type 19 (line 12). The `send()` method in Fig. 1.5 records the handler object (an object of the `ViewRootHandler` class) and the message type (represented by integer constant 19) with the message object. After dequeuing the message, the looper invokes the `handle()` method of the specified handler to dispatch the message (line 4 in Fig. 1.5).

```
public class Message {
  Handler target;
  int what;
}
public class MessageQueue {
  void enqueue(Message m) {...}
  Message next() {...}
}
```

(a) `Message` and `MessageQueue` interfaces.

```
abstract class Handler {
  MessageQueue mQueue;
  void send(int w) {
    Message m = new Message();
    m.target = this;
    m.what = w;
    mQueue.enqueue(m);
  }
  abstract void handle(Message m);
}
public class Looper {
  MessageQueue mQueue;
  void loop() {
    for (;;) {
      Message m = mQueue.next();
      Handler h = m.target;
      h.handle(m);
    }
  }
}
```

(b) Message enqueuing and dispatching framework.

Fig. 1.5.: Messaging framework of Android.

```
public class ViewRootHandler extends Handler {
  public void handle(Message m) {
    int w = m.what;
    switch (w) {
    case 19: ... // handling
    }
  }
}
public class ViewRootImpl {
  ViewRootHandler mHandler;
  private void schedule() {
    mHandler.send(19); // sending
  }
}
```

Fig. 1.6.: Message-handling in Android framework.

We show that our demand-driven approach to analysis can identify and focus on part of the program related to message-passing – making message-driven programs easier to understand and debug (Chapter 8).

The rest of this thesis is organized as follows. Chapter 2 specifies a bytecode-like example language to present code examples in this dissertation. Chapter 3 surveys four representative heap analyses to outline the evolutionary path of today's most heap analyses. Chapter 4 presents Clipper – an access-path based on-demand heap analysis to resolve heap-carried data flows. Chapters 5 to 8 describe applications of Clipper to solve three practical problems in analyses of large scale programs: resolving implicit control flows introduced by callback mechanism, demand-driven refinement of points-to analysis, and resolving asynchronous control flows introduced by message-passing.

2. EXAMPLE LANGUAGE

We explain our ideas with a simple Java-bytecode-like language (defined in Fig. 2.1) in which a program is a set of labeled statements $\overline{stmt}$. For the rest of this dissertation we assume the declaration of p is "$t\ p(t_0\ h_0, ..., t_k\ h_k)\{body_p\}$".

class name $t \in Class$	object field name $f, g \in OField$
method name $p, q \in Method$	static field name $\mathbb{f}, \mathbb{g} \in SField$
variable name $x, y, z \in Var$	formal parameter name $h \in Param \subseteq Var$
statement label $l \in Label = \mathbb{N}$	

$prog$	::=	$\overline{cdecl}$	// program
$cdecl$	::=	class t $\{\overline{fdecl}\ \overline{mdecl}\}$	// class declaration
$fdecl$	::=	$t\ f$	// field declaration
$mdecl$	::=	$t\ p(\overline{t\ h})\ \{body\}$	// method declaration
$body$	::=	$\overline{stmt}$	// method body
$stmt$	::=	l: x = new t | l: $x = y$ |	// allocation and assignment
		l: $x = y.f$ | l: $x.f = y$ |	// object field load and store
		l: $x = \mathbb{f}$ | l: $\mathbb{f} = x$ |	// static field load and store
		l: goto l' | l: if $b\ l_t\ l_f$	// branches
		l: $x = p(\overline{y})$ | l: return x	// method call and return
		l^{x}: exit	// pseudo method exit

Fig. 2.1.: Syntax of a simple bytecode-like language.

3. EVOLUTION OF HEAP ANALYSIS TECHNIQUES

3.0.1 Global-Heap-Based Approach

Many analyses resolve heap-carried dependency with global heap models [12]. We now describe how global-heap-based approaches are limited by their consideration of unrealizable paths which can lead, e.g., to conservatism in detecting downcast failures.

Typical global heap models include points-to graphs generated by points-to analyses. Within a points-to graph, objects are modeled with global names such as Vec_{name} and Vec_{Int} denoting the vectors allocated at line 17 and 27, respectively, and Arr denoting the array allocated at line 4. As an example, Fig. 3.1 shows part of the data flow generated by a context-insensitive points-to analysis of the example program in Fig. 1.1. Due to the absence of context, the flows of Vec_{name} and Vec_{Int} merge along paths $a \rightarrow c$ and $b \rightarrow c$ respectively, where their own arrays, denoted by the common name Arr, are modeled as being stored to their `arr` field, as shown in the generated points-to graph in the bottom-right corner of Fig. 3.1. Similarly, due to the flow confluence of these two vectors along paths $f \rightarrow g \rightarrow l$ and $h \rightarrow j \rightarrow l$, and the flow confluence of the string $name$ and integer 3 along paths $e \rightarrow g \rightarrow k$ and $i \rightarrow j \rightarrow k$, both $name$ and 3 are modeled as being stored to Arr at line 11, as shown in the generated points-to graph.

With the heap modeled globally as a points-to graph, points-to analyses propagate points-to relations globally, generating global data flows between interfering heap loads/stores, i.e. loads/stores on the same object. As shown in Fig. 3.2, since Vec_{Int} flows to `Vector.get(i)` along the path $n \rightarrow o \rightarrow p$, the array Arr referenced by $Vec_{Int}.arr$ is loaded at line 8. Hence $name$ and 3, which are stored to the same Arr at line 11, propagate globally along the arrow z. These values are further returned to the call site at line 30, causing a (false) downcast failure.

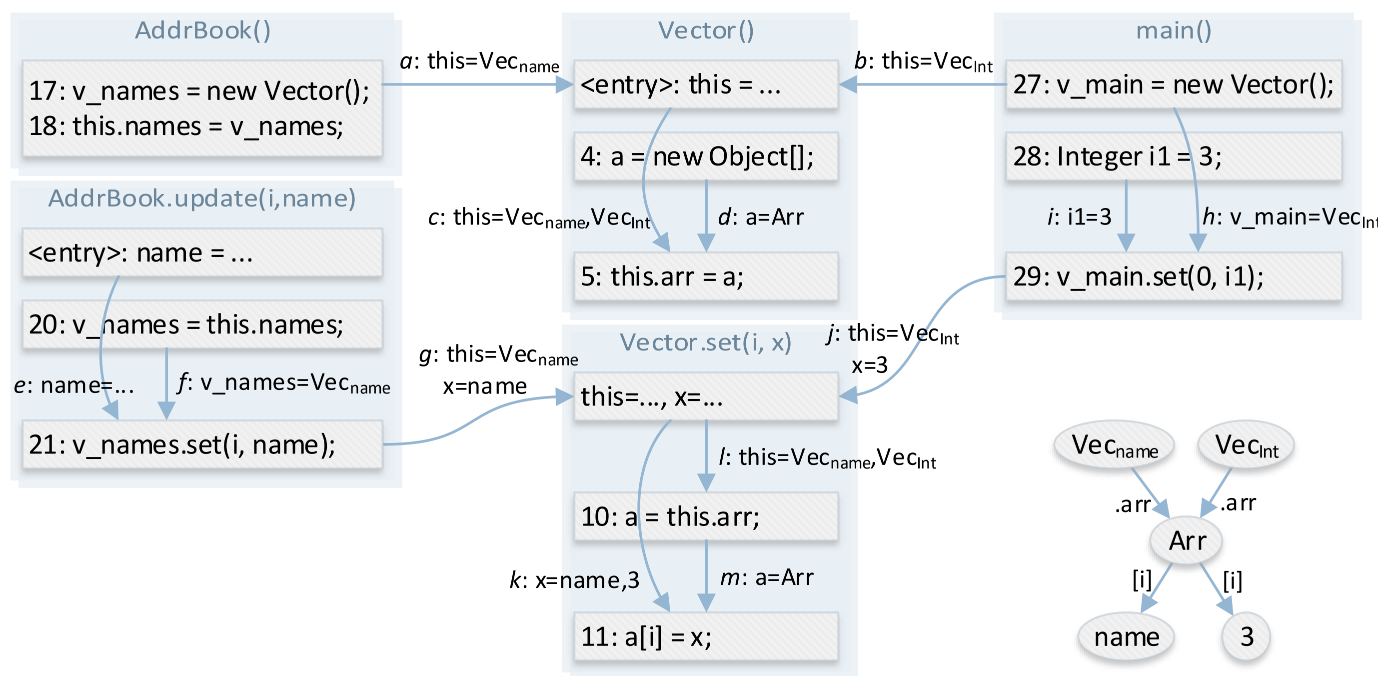


Fig. 3.1.: Part of the data flows of a context-insensitive points-to analysis of the example program (Fig. 1.1). Numbers in boxes are the line number from Fig. 1.1. Part of the generated points-to graph is shown in the bottom-right corner.

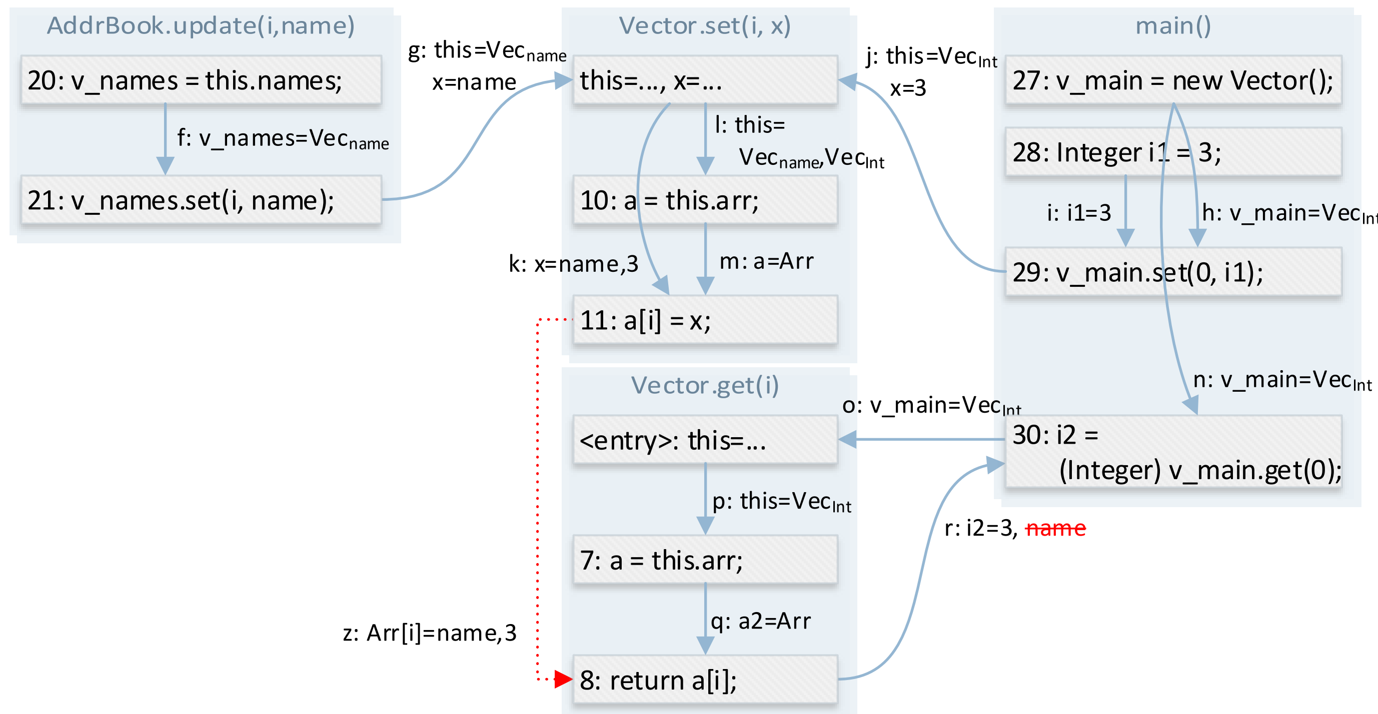


Fig. 3.2.: An example *global* data flow (the red dotted arrow) between interfering load/store on Arr.

In an attempt to identify the data flows relevent to the returned value at line 30, several derivation-based approaches [13, 14] are developed. Such approaches essentially back-trace data flows derived by the points-to analysis, including these global ones. However, valuable context information is lost when back-tracing these global flows. For example, assume we back-trace the data flow reaching the cast at line 30. Along the path $r \leftarrow z \leftarrow m \leftarrow l$ in Fig. 3.2), the tracing reaches the parameter `this` of `Vector.set`(). Because the tracing propagates to `Vector.set`() along the global flow o, there is no context information from the trace indicating which call site the tracing should further propagate up to. Therefore, the tracing has to conservatively propagate to all call sites, including the one at line 21 (along arrow g), which is actually irrelevant to the data flow reaching the cast at line 30. Although a finer points-to graph generated by a context-sensitive points-to analysis can avoid such spurious tracing in this example, such solutions can only mitigate the problem rather than eliminate it, while incurring high overhead.

When tracing interprocedurally, contexts contain valuable information recording the call sites the tracing was triggered by and will return to. Thus many analyses try to preserve this information by applying a local heap model [12] where there is no global data flow. One such analysis is the context-free-language reachability analysis [5] where the heap is modeled with a context-free language and the context is modeled with call strings.

3.0.2 The *CFL* Approach

We now describe how the context-free-language reachability analysis (*CFL*) [5] with call-string-based context-sensitivity is limited by its need to truncate call strings in the presence of recursion, which incurs spurious data flows.

In *CFL*, a bidirectional data flow is modeled with a string s which is a mixture of two substrings s_F and s_C where s_F represents part of the flow through the heap and s_C represents the interprocedural part of the flow. For example, the bidirectional data flow from variable $i2$ in Fig. 3.3 is modeled by the string s = "$(_{30} \cdot [_{[i]} \cdot [_{arr} \cdot)_{30} \cdot (_{29} \cdot]_{arr} \cdot]_{[i]} \cdot)_{29}$" where s_F = "$[_{[i]} \cdot [_{arr} \cdot]_{arr} \cdot]_{[i]}$" and s_C = "$(_{30} \cdot)_{30} \cdot (_{29} \cdot)_{29}$". The symbol "$(_{30}$" at arrow a represents a downward data flow from caller to callee at the call site of line 30. Correspondingly the symbol "$)_{30}$" at arrow d represents an upward data flow from callee to caller at the call site of line 30. The symbol "$[_{arr}$" at arrow c represents the alias relation between expressions a and $this.\boldsymbol{arr}$ at line 7. Correspondingly "$]_{arr}$" at arrow f represents the alias relation between expressions $this.\boldsymbol{arr}$ and a at line 10.

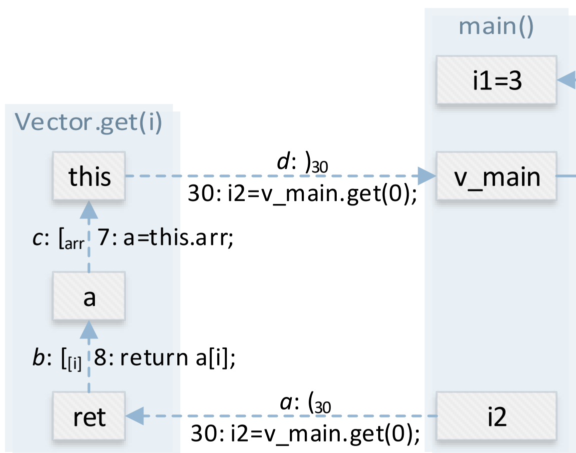


Fig. 3.3.: Example of realizable data flow from $i2$. The nodes represent local variable values[1]. Solid/dashed arrows represent forward/backward transitions between values incurred by the labeled statements.

The combined data flow is realizable only if $s_F \in L_F$ and $s_C \in L_C$ where L_F and L_C are two context-free languages defined by the following grammars.

$$L_F \rightarrow [_f \ L_F \ ]_f \mid [_g \ L_F \ ]_g \mid \ldots \mid \epsilon \qquad \text{where } f, g \in \mathit{Field}$$

$$L_C \rightarrow (_i \ L_C \ )_i \mid (_j \ L_C \ )_j \mid \ldots \mid \epsilon \qquad \text{where } i \text{ and } j \text{ are call sites}$$

In Fig. 3.3, for example, the string s of the flow consists of s_C = "$(_{30} \cdot)_{30} \cdot (_{29} \cdot)_{29}$" $\in L_C$ and s_F = "$[_{[i]} \cdot [_{arr} \cdot]_{arr} \cdot]_{[i]}$" $\in L_F$. Hence the flow is realizable.

[1]Array elements are modeled with a pseudo field denoted by $[i]$ and return statements are modeled with assignments to the pseudo variable *ret*.

A variable may point to certain value if there is a realizable data flow between them. In Fig. 3.3, for example, $i2$ may point to “3” because of the realizable data flow. Given a variable x as a query, the *CFL* analysis returns all x's possible values reachable via realizable data flows.

The language L_C essentially encodes contexts with call-strings. As a top-down approach to interprocedural analysis, call-string-based context representation requires truncation [15] to stay bounded in the presence of recursive method calls, and this truncation leads to precision loss. Consider the example in Fig. 3.4. Because of truncation, the recursive call in the method has the same effect as `GOTO`, i.e., jumping to the entry of `foo()` without extending the call string, and generates spurious data flows between x and y.

```
void foo(T x, T y, int c) {
  if (c == 0) return;
  foo(y, x, c--); // swap x and y
}
```

Fig. 3.4.: Example code for recursive invocation.

To avoid precision loss from truncation, many interprocedural analyses are summary-based (also known as *functional* in [16]) rather than call-string based. One such approach is FlowDroid [17] – a popular taint analysis tool.

3.0.3 Bottom-Up Summary-Based Approach

We next show bottom-up built method summaries contain excessive side-effects, that is irrelevant to the flow of interest and incurs unnecessary analysis cost. To replace call-string based approach with summary-based one, local heap needs to be modeled separately from contexts, unlike *CFL* where the sub-string modeling heap and that modeling context are mixed in the data flow. In [18] and many other summary-based analysis, local heap is modeled with access paths where each access path α is a local variable x followed by a field path δ (written $x.\delta$) and each field path δ is a (potentially empty) sequence of field names, as defined below:

(concrete) field path $\delta \in \Delta = OField^*$ // The empty path is denoted by ϵ

(concrete) access path $\alpha, \beta, x.\delta \in AP = Var \times \Delta$

concatenation $\cdot . \cdot : \Delta \times \Delta \rightarrow \Delta$ s.t. $\langle f_1, ..., f_m \rangle . \langle g_1, ..., g_n \rangle \triangleq \langle f_1, ..., f_m, g_1, ..., g_n \rangle$

concatenation $\cdot . \cdot : AP \times \Delta \rightarrow AP$ s.t. $\langle r, \delta \rangle . \delta' \triangleq \langle r, \delta . \delta' \rangle$

An access path (e.g. $this.arr[i]$) represents the memory location to which the access path evaluates as an expression at runtime. The approach in [18] builds side-effect summary of each method describing its read and write sets in terms of access paths rooted in the method's parameters and returned values as shown in Table 3.1. Then there is no need to propagate callers' data flow down to callees, as a callee's side-effects are modeled with its summary, and no need to model contexts with call-strings.

Table 3.1.: Side-effect summaries of `Vector`'s methods.

Method	Read Set	Write Set
`Vector()`	$\emptyset$	$\{this.arr\}$
`Vector.get(i)`	$\{this.arr, this.arr[i]\}$	$\emptyset$
`Vector.set(i,x)`	$\{this.modCount, this.arr\}$	$\{this.modCount, this.arr[i]\}$

However, by building a callee's side-effect summary in a solely bottom-up way, the summary has to conservatively include all side-effects of the callee, including those uninteresting to callers. For example, the access path $this.modCount$ in read/write sets of `Vector.set(i,x)` is irrelevant to the flow of $i1$ at the call site of line 29. To avoid including irrelevant information in the callee's summary, on-demand building of callee's summary has been proposed. One such approach is FLOWDROID [17] – a popular security analysis tool based on taint analysis.

3.0.4 The FlowDroid Approach

Next, we discuss FlowDroid – a taint analysis tool implementing an on-demand summary-based approach to heap analysis. To replace the call-string based approach with a summary-based one, the heap needs to be modeled separately from contexts, unlike *CFL* where the sub-strings modeling heap and that modeling context are mixed in with the data flow. In FlowDroid and many other summary-based analyses, the heap is modeled with access paths where each access path α is a local variable x followed by a field path δ (written $x.\delta$) and each field path δ is a (potentially empty) sequence of field names, as defined below:

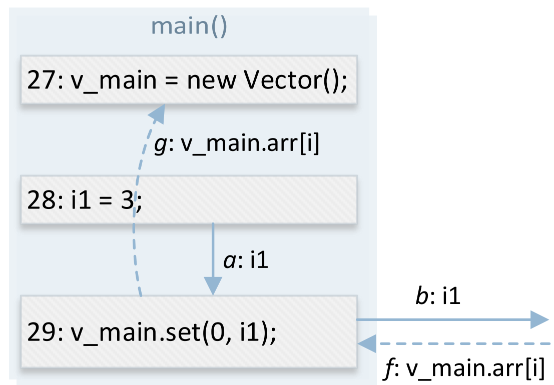
$$\text{concrete field path } \delta \in \Delta = OField^* \text{ // The empty path is denoted by } \epsilon$$

$$\text{concrete access path } \alpha, \beta, x.\delta \in AP = Var \times \Delta$$

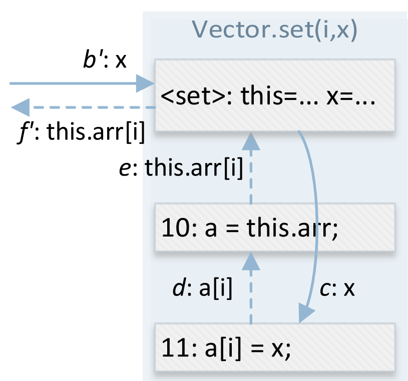
An access path (e.g. $this.arr[i]$) represents the value to which the path evaluates as an expression at runtime. Given certain method, FlowDroid builds a summary of it describing tainted output values given certain tainted input value. Both tainted input and output values are represented by access paths rooted in the method's parameters or returned value. In Fig. 3.5b for example, given a tainted input value x at arrow b', the execution of `set()` taints the value $this.arr[i]$ as an output (arrow f'). The meaning of the data flows is given in Chapter 4.

Method summaries are context-free, which means a method summary can be universally applied to any call site of the method, like a function, hence *functional*. For example, at the call site of line 29 where $i1$ is tainted (arrow a in Fig. 3.5a), applying the summary of `set()` gives us another tainted value $v\_main.arr[i]$ at arrow f. Thus there is no need to maintain contexts when building a method's summary and no need to model contexts with call-strings.

Given a tainted source value as a query, FlowDroid returns all tainted sinks as potential leaks. There is, however, no existing formalization of the semantics underlying FlowDroid in the literature to verify its soundness. We now formulate



(a) Data flow with respect to the query $i1$.



(b) Data flow with respect to the tainted input value x.

Fig. 3.5.: Example illustrating FlowDroid. The solid/dashed arrows are explained in Chapter 4.

one below and use it to expose a loophole in the design of FlowDroid as a security analysis tool (Section 4.0.3).

4. CLIPPER – A NEW ON-DEMAND HEAP ANALYSIS

4.0.1 A Denotational Semantics for Procedure Alias Effect

With a forward taint analysis and a backward on-demand alias analysis integrated [17], FlowDroid essentially implements a *bidirectional* analysis capable of collecting all alias access paths referring to the tracked value.

Given a semantic domain consisting of all alias classes where an alias class A is a set of access paths alias with each other, as defined below:

$$\begin{aligned}
&\text{alias class } A \in AClass = 2^{AP} \\
&\text{concatenation } \ \cdot . \cdot : AClass \times \Delta \to AClass \text{ s.t. } A.\delta \triangleq \{\alpha.\delta \mid \alpha \in A\} \\
&\text{concatenation } \ \cdot . \cdot : AClass \times 2^{\Delta} \to AClass \text{ s.t. } A.D \triangleq \{\alpha.\delta \mid \alpha \in A \wedge \delta \in D\}
\end{aligned}$$

The denotational semantics of intraprocedural statements is defined below, where the *effect* of each statement *stmt* is denoted by a function $[\![stmt]\!]^{\mathbb{A}} \in AClass \to AClass$ extending an alias class A (hence the "$A \cup$" component) with new may-alias access paths implied by that statement.

$$\begin{aligned}
[\![l\colon x = y]\!]^{\mathbb{A}}(A) &:= A \cup \{y.\delta \mid x.\delta \in A\} \cup \{x.\delta \mid y.\delta \in A\} \\
[\![l\colon x = y.f]\!]^{\mathbb{A}}(A) &:= A \cup \{y.\delta_f \mid x.\delta \in A\} \cup \bigcup_{\delta' \in \delta_{\overline{f}}} \{x.\delta' \mid y.\delta \in A\} \\
[\![l\colon y.f = x]\!]^{\mathbb{A}}(A) &:= A \cup \{y.\delta_f \mid x.\delta \in A\} \cup \bigcup_{\delta' \in \delta_{\overline{f}}} \{x.\delta' \mid y.\delta \in A\} \\
&\text{where } \delta_f := f.\delta \text{ and } \delta_{\overline{f}} := \begin{cases} \{\delta'\} & \text{if } \delta = f.\delta' \\ \emptyset & \text{otherwise} \end{cases}
\end{aligned}$$

The function δ_f "cons" the field f to the field path δ (e.g. $(arr[i])_{names} = names.arr[i]$) and the inverse function $\delta_{\overline{f}}$ returns the tail of the field path δ if its head matches the field f (e.g. $(names.arr[i])_{\overline{names}} = \{arr[i]\}$). The semantic functions symmetrically apply to access paths rooted at both left- and right-hand sides of the statements, essentially renders the semantics bidirectional (or *flow-insensitive*).

The effect summary of a method body, e.g. $[\![body]\!]^{\mathbb{A}} \in AClass \rightarrow AClass$, is defined by repeatedly applying the extending function $[\![stmt]\!]^{\mathbb{A}}$ of each statement $stmt \in body$, i.e.,

$$[\![body]\!]^{\mathbb{A}} := \mathsf{FIX}\ \lambda d.\ \lambda A.\ A \cup \bigcup_{stmt_i \in body} d \circ [\![stmt_i]\!]^{\mathbb{A}}(A)$$

Example 1 *Given an initial alias class* A=$\{ret_{get}\}$ *within method* **`Vector.get()`**, *the execution of "***`return a[i]`***" generates:*[1]

$$A_1 {=} [\![return\ a[i]]\!]^{\mathbb{A}}(A) {=} [\![ret_{get}{=}a[i]]\!]^{\mathbb{A}}(A) {=} \{ret_{get}, a[i]\}$$

*Similarly the execution of "***`a=this.arr`***" further generates:*

$$A_2 {=} [\![a{=}this.arr]\!]^{\mathbb{A}}(A_1) {=} \{ret_{get}, a[i], this_{get}.arr[i]\}$$

Since further iterations add no new alias access paths, the effect summary of executing the method is the extended alias class $[\![body_{get}]\!]^{\mathbb{A}}(A) {=} \{ret_{get}, a[i], this_{get}.arr[i]\}$.

For any statement "l: $x{=}p(y_0, \ldots, y_k)$" where the declaration of p is

$$t\ p(t_0\ h_0, \ldots, t_k\ h_k)\{body_p\}$$

[1]method returns are modeled as assignments to variable "ret"

the following domains and helper functions are defined:

$$\text{downward visible access paths } AP_l^{\downarrow} = x.\Delta \ \cup \bigcup_{0 \le i \le k} y_i.\Delta$$

$$\text{upward visible access paths } AP_p^{\uparrow} = ret_p.\Delta \ \cup \bigcup_{0 \le i \le k} h_i.\Delta$$

$$\text{downward mapping } \cdot{\downarrow_p^l} \colon AP_l^{\downarrow} \to AP_p^{\uparrow}$$

$$\text{s.t. } \forall \delta \in \Delta : x.\delta{\downarrow_p^l} = ret_p.\delta \wedge \forall i \in [0,k] : y_i.\delta{\downarrow_p^l} = h_i.\delta$$

$$\text{upward mapping } \cdot{\uparrow_p^l} \colon AP_p^{\uparrow} \to AP_l^{\downarrow}$$

$$\text{s.t. } \forall \delta \in \Delta : ret_p.\delta{\downarrow_p^l} = x.\delta \wedge \forall i \in [0,k] : h_i.\delta{\downarrow_p^l} = y_i.\delta$$

$$\text{downward mapping } \cdot{\downarrow_p^l} \colon AClass \to AClass$$

$$\text{s.t. } \forall A \in AClass : A{\downarrow_p^l} = map^2(\cdot{\downarrow_p^l})(A \cap AP_l^{\downarrow})$$

$$\text{upward mapping } \cdot{\uparrow_p^l} \colon AClass \to AClass$$

$$\text{s.t. } \forall A \in AClass : A{\uparrow_p^l} = map(\cdot{\uparrow_p^l})(A \cap AP_p^{\uparrow})$$

A method call $x{=}p(\overline{y})$ denotes a function extending the caller's alias class A with alias access paths implied by the body of the callee $body_p$:

$$[\![l \colon x{=}p(\overline{y})]\!]^{\mathbb{A}}(A) := A \cup ([\![body_p]\!]^{\mathbb{A}}(A{\downarrow_p^l})){\uparrow_p^l}$$

Given a caller alias class A at call site $x{=}p(\overline{y})$, $A{\downarrow}$ collects a subset of A visible to callee (i.e., those access paths rooted at y_i and x) and maps the subset to callee's scope (i.e., mapping $y_i.\delta$ to $h_i.\delta$ and $x.\delta$ to $ret_p.\delta$). Then $[\![body_p]\!]^{\mathbb{A}}(A{\downarrow})$ returns the effect summary of callee – an alias class generated by extending $A{\downarrow}$. Then $([\![body_p]\!]^{\mathbb{A}}(A{\downarrow})){\uparrow}$ instantiates this callee effect by collects a subset of it visible to the caller (i.e., those access paths rooted at h_i and ret_p) and maps the subset to caller's scope (i.e., mapping $h_i.\delta$ to $y_i.\delta$ and $ret_p.\delta$ to $x.\delta$) – a process similar to *effect masking* [19].

[2] where $map(f)(M) \triangleq \{f(x) \mid x \in M\}$

Example 2 *Given the caller alias class $A = \{i2\}$ at line 30 in our example (thus $A{\downarrow} = \{ret_{get}\}$), the effect summary of the callee is*

$$[\![body_{get}]\!]^{\mathbb{A}}(A{\downarrow}) = \{ret_{get}, a[i], this_{get}.arr[i]\}$$

as shown in Example 1, and the instantiation of the effect summary is $([\![body_{get}]\!]^{\mathbb{A}}(A{\downarrow}))\uparrow = \{i2, v\_main.arr[i]\}$.

In the next section, we propose Clipper – a new on-demand heap analysis based on this semantics. We then show a loophole in the design of FlowDroid by comparing it with Clipper.

4.0.2 A Specification based on Deduction Rules

In this section, a new on-demand heap analysis – Clipper – is specified with a set of deduction rules encoding FlowDroid's semantics (Chapter 4). A loophole that undermine FlowDroid's soundness as a security analysis tool is exposed by comparing FlowDroid with Clipper (Section 4.0.3). Handling of static fields and its generalization are introduced in Section 4.0.4.

Similar to an equivalence class, an alias class can be represented by one of its members – its *representative*. Thus the alias class containing an access path α is denoted by $[\alpha]$. The membership relation $\beta \in [\alpha]$ is encoded by the fact $A(\alpha, \beta)$, and thus the fact $A(\alpha, \alpha)$ holds trivially. According to the semantics defined in Section 4.0.1, an on-demand heap analysis can be specified with deduction rules for inferring the alias class $[\alpha]$ of certain access path α given as a *query*, i.e., deriving all facts of the form "$A(\alpha, \_)$". We call this rule-based on-demand heap analysis Clipper.

In Clipper, program elements are encoded with a set of *base facts* defined in Fig. 4.1 and a query α is encoded as the initial (trivially holding) fact $A(\alpha, \alpha)$.

$$
\begin{aligned}
l\colon x = y &\Rightarrow \mathit{Assign}(l, x, y) & l\colon x = \text{new } t &\Rightarrow \mathit{Alloc}(l, x)\\
l\colon x = y.f &\Rightarrow \mathit{OLoad}(l, x, y, f) & l\colon x.f = y &\Rightarrow \mathit{OStore}(l, x, f, y)\\
l\colon x = \mathbb{f} &\Rightarrow \mathit{SLoad}(l, x, \mathbb{f}) & l\colon \mathbb{f} = x &\Rightarrow \mathit{SStore}(l, \mathbb{f}, x)
\end{aligned}
$$

$$
l\colon x{=}p(y_0, \ldots, y_k) \Rightarrow \begin{cases} \mathit{VCall}(l, y_0, p) & \text{// virtual call on receiver } y_0\\ \mathit{CallArg}(l, i, y_i) \text{ for any } i \in [0, k] & \text{// } i\text{th argument is } y_i\\ \mathit{CallRet}(l, x) & \text{// call returns value to } x \end{cases}
$$

$$
l\colon \text{return } z \text{ in method } p \Rightarrow \mathit{Assign}(l, ret_p, z) \text{ // method returns are modeled as assignments to variable ``}\mathit{ret}\text{''}
$$

$$
t\ p(t_0\ h_0, \ldots, t_k\ h_k)\{body_p\} \Rightarrow \mathit{Param}(p, i, h_i) \text{ for any } i \in [0, k]
$$

Fig. 4.1.: The base facts represented by the program elements.

For virtual calls, the first argument y_0 denotes the receiver one.

The intraprocedural semantic functions can be encoded with the following rules.

$$[\![l\colon x = y]\!]^{\mathbb{A}} \Rightarrow \begin{cases} A(\alpha, y.\delta) \text{ :- } A(\alpha, x.\delta), Assign(l, x, y). \\ A(\alpha, x.\delta) \text{ :- } A(\alpha, y.\delta), Assign(l, x, y). \end{cases} \tag{ASSIGN}$$

$$[\![l\colon x = y.f]\!]^{\mathbb{A}} \Rightarrow \begin{cases} A(\alpha, y.\delta_f) \text{ :- } A(\alpha, x.\delta), OLoad(l, x, y, f). \\ A(\alpha, x.\delta') \text{ :- } A(\alpha, y.\delta), OLoad(l, x, y, f), \delta' \in \delta_{\overline{f}}. \end{cases} \tag{OLOAD}$$

$$[\![l\colon y.f = x]\!]^{\mathbb{A}} \Rightarrow \begin{cases} A(\alpha, y.\delta_f) \text{ :- } A(\alpha, x.\delta), OStore(l, y, f, x). \\ A(\alpha, x.\delta') \text{ :- } A(\alpha, y.\delta), OStore(l, y, f, x), \delta' \in \delta_{\overline{f}}. \end{cases} \tag{OSTORE}$$

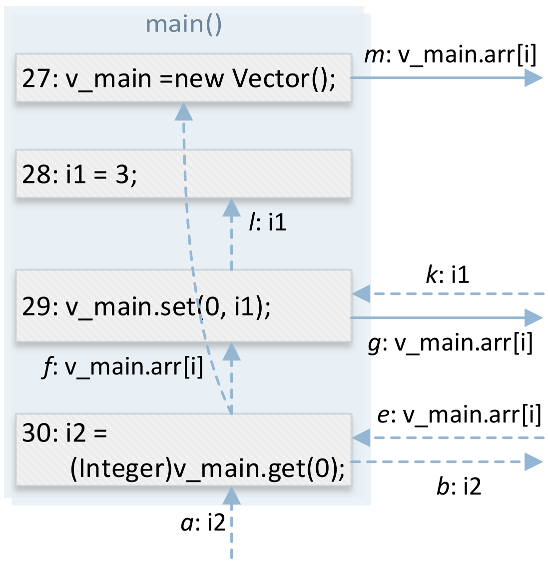
Example 3 *The derivation corresponding to the evaluation in Example 1 is shown in Fig. 4.4b, which can be visualized as data flows in Fig. 4.3b, with each statement representing the firing of the corresponding rule and the incoming/outgoing flows representing the condition/consequent alias facts of the firing. For example, the initial alias fact $A(ret, ret)$ triggers the firing of rule OLOAD at line 8, generating $A(ret, a[i])$ at arrow c. Similarly, the new fact triggers the firing of rule OLOAD again at line 7, generating $A(ret, this.arr[i])$ at arrow d. Another derivation for the alias of $this.arr[i]$ in* `Vector.set()` *is visualized in Fig. 4.3a. As shown in the figure, derivations are essentially bidirectional.*

The encoding of interprocedural semantics with rules entails a more theoretical construction.

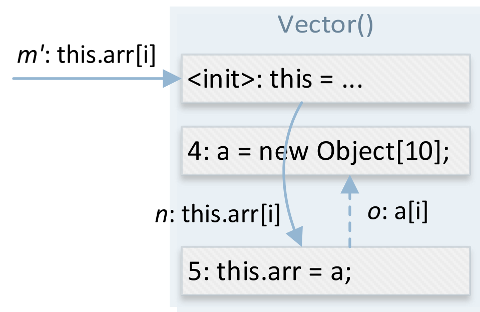
Theorem 4.0.1 *Given the complete lattice $(L, \sqsubseteq, \sqcup, \sqcap, \perp, \top)$ where $L = AClass$, $\sqsubseteq$ is $\subseteq$, $\sqcup$ is $\cup$, $\sqcap$ is $\cap$, $\perp = \emptyset$, $\top = AP$, we have*

1. *$\forall stmt$: $[\![stmt]\!]^{\mathbb{A}}$ is completely additive, i.e.,*

$$\forall \mathcal{A} \subseteq AClass\colon [\![stmt]\!]^{\mathbb{A}}(\bigsqcup \mathcal{A}) = \bigsqcup_{A \in \mathcal{A}} \{[\![stmt]\!]^{\mathbb{A}}(A)\}$$

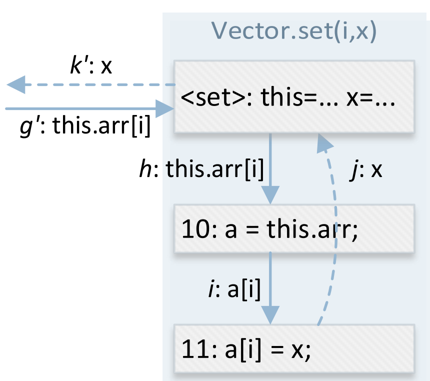


(a) The derivation for facts of the form $A(i2, \_)$.

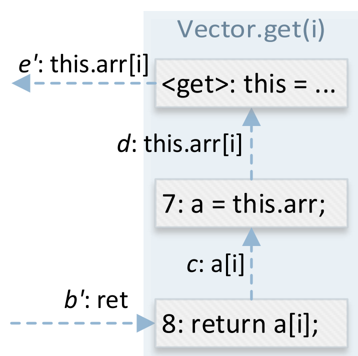


(b) The derivation for facts of the form $A(this.arr[i], \_)$.

Fig. 4.2.: Visualizing derivations as data flows. The derived facts $A(\alpha, X)$ are visualized as arrows labeled by X. Solid and dashed arrows are forward and backward data flows, respectively.



(a) The derivation for facts of the form $A(this.arr[i], \_)$



(b) The derivation for facts of the form $A(ret, \_)$

Fig. 4.3.: Visualizing derivations as data flows. The derived facts $A(\alpha, X)$ are visualized as arrows labeled by X. Solid and dashed arrows are forward and backward data flows, respectively.

2. $\forall body$*: $[\![body]\!]^{\mathbb{A}}$ is completely additive, i.e.,*

$$\forall \mathcal{A} \subseteq AClass\colon [\![body]\!]^{\mathbb{A}}(\bigsqcup \mathcal{A}) = \bigsqcup_{A \in \mathcal{A}} \{[\![body]\!]^{\mathbb{A}}(A)\}$$

Proof For intraprocedural statements, (1) can be proved via a case-by-case analysis of the semantic function. For method calls, the proofs of (1) and (2) depend on each other, hence needs to proceed by coinduction [20].

Assuming (1) holds, the proof of (2) is similar to the one of Lemma 5.44 in [21]. We may first prove $H^n\bot$ is completely additive, then $\mathsf{FIX}\ H = \sqcup\{H^n\bot \mid n \geq 0\}$ is completely additive. Then assuming (2) holds, it follows that

$$\begin{aligned}
[\![l\colon x{=}p(\overline{y})]\!]^{\mathbb{A}}(\cup A_i) &= (\cup A_i) \cup ([\![body_p]\!]^{\mathbb{A}}((\cup A_i)\downarrow))\uparrow \\
&= (\cup A_i) \cup (\cup[\![body_p]\!]^{\mathbb{A}}(A_i\downarrow))\uparrow \\
&= (\cup A_i) \cup \bigcup([\![body_p]\!]^{\mathbb{A}}(A_i\downarrow))\uparrow \\
&= \bigcup A_i \cup ([\![body_p]\!]^{\mathbb{A}}(A_i\downarrow))\uparrow \\
&= \ \cup\, [\![x{=}p(\overline{y})]\!]^{\mathbb{A}}(A_i)
\end{aligned}$$

This completes the proof. ■

Given the complete additivity of $[\![body_p]\!]^{\mathbb{A}}$ (Theorem 4.0.1), it follows that

$$
\begin{aligned}
& [\![l\colon x{=}p(\overline{y})]\!]^{\mathbb{A}}(A) \\
:= {} & A \cup ([\![body_p]\!]^{\mathbb{A}}(A\!\downarrow))\!\uparrow \\
= {} & A \cup ([\![body_p]\!]^{\mathbb{A}}(\{h_i.\delta \mid y_i.\delta \in A\} \cup \{ret_p.\delta \mid x.\delta \in A\}))\!\uparrow \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(by definition of } A\!\downarrow) \\
= {} & A \cup (\bigcup_{y_i.\delta \in A} [\![body_p]\!]^{\mathbb{A}}(\{h_i.\delta\}) \cup \bigcup_{x.\delta \in A} [\![body_p]\!]^{\mathbb{A}}(\{ret_p.\delta\}))\!\uparrow \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(by additivity of } [\![body_p]\!]^{\mathbb{A}}) \\
= {} & A \cup \begin{cases}
\{y_j.\delta_j \mid y_i.\delta_i \in A \wedge h_j.\delta_j \in [\![body_p]\!]^{\mathbb{A}}(\{h_i.\delta_i\})\} & (a) \\
\{x.\delta \mid y_i.\delta_i \in A \wedge ret_p.\delta \in [\![body_p]\!]^{\mathbb{A}}(\{h_i.\delta_i\})\} & (b) \\
\{y_i.\delta_i \mid x.\delta \in A \wedge h_i.\delta_i \in [\![body_p]\!]^{\mathbb{A}}(\{ret_p.\delta\})\} & (c) \\
\{x.\delta_2 \mid x.\delta_1 \in A \wedge ret_p.\delta_2 \in [\![body_p]\!]^{\mathbb{A}}(\{ret_p.\delta_1\})\} & (d)
\end{cases} \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(by definition of } A\!\uparrow)
\end{aligned}
$$

Each callee-visible access path is a new query to the callee and there are two kinds – those rooted at parameters h_i (cases a and b), and those rooted at returned value ret_p (cases c and d). For each query there are two kinds of caller-visible access paths in the callee effect summary – (1) those rooted at parameters h_i (cases a and c); and

(2) those rooted at returned value ret_p (cases b and d). There are four cases in total, which are encoded correspondingly with the following rules:

$$
[\![l\colon x{=}p(\overline{y})]\!]^{\mathbb{A}} \Rightarrow
\begin{cases}
A(\alpha, y_j.\delta_j) \text{ :- } A(\alpha, y_i.\delta_i), CallArg(l, i, y_i), CallArg(l, j, y_j), \\
\qquad Param(p, i, h_i), Param(p, j, h_j), A(h_i.\delta_i, h_j.\delta_j). & (a) \\
A(\alpha, x.\delta) \text{ :- } A(\alpha, y_i.\delta_i), CallArg(l, i, y_i), CallRet(l, x), \\
\qquad Param(p, i, h_i), A(h_i.\delta_i, ret_p.\delta). & (b) \\
A(\alpha, y_i.\delta_i) \text{ :- } A(\alpha, x.\delta), CallRet(l, x), CallArg(l, i, y_i), \\
\qquad Param(p, i, h_i), A(ret_p.\delta, h_i.\delta_i). & (c) \\
A(\alpha, x.\delta_2) \text{ :- } A(\alpha, x.\delta_1), CallRet(l, x), A(ret_p.\delta_1, ret_p.\delta_2). & (d)
\end{cases}
\quad \text{(CALL)}
$$

Observation All derived facts are of the form "$A(\alpha, \_)$" and all rules are guarded by existing facts of the form $A(\alpha, \_)$ where the first bound argument α is the query to answer.

Such facts serve a similar role as *magic facts* implying that "The problem of determining alias class of α arises" [3]. Given this, the analysis may leverage the *magic-sets* approach [3] to build method effect summaries (i.e., alias classes) on demand, by deriving facts to answer certain query only. Given a query "α", the initial magic fact $A(\alpha, \alpha)$ is added to the derivation. Additional magic facts initiating analysis on callees are generated by following rules:

$$
l\colon x{=}p(\overline{y}) \Rightarrow
\begin{cases}
A(h_i.\delta_i, h_i.\delta_i) \text{ :- } A(\alpha, y_i.\delta_i), CallArg(l, i, y_i), Param(p, i, h_i). & (e) \\
A(ret_p.\delta, ret_p.\delta) \text{ :- } A(\alpha, x.\delta), CallRet(l, x). & (f)
\end{cases}
\quad \text{(CALL)}
$$

Example 4 *In the example program, the caller fact $A(i2, i2)$ initiates a magic fact $A(ret, ret)$ on callee* `Vector.get()` *at line 30 by rule CALL(f), as shown by the derivation in Fig. 4.4a which is visualized by arrows b and b' in Fig. 4.2. This callee fact further triggers the derivation of a callee effect summary $A(ret, this.arr[i])$,* *as shown in Fig. 4.3b. Then rule CALL(c) instantiates the summary and further derives the caller fact $A(i2, v\_main.arr[i])$ as shown by the derivation in Fig. 4.4c and visualized by arrow e in Fig. 4.2a. According to rule CALL(e), this caller fact further triggers the analysis of* `Vector.set()` *at line 29, as visualized by the arrow g'. The derived callee summary $A(this.arr[i], x)$ (Fig. 4.3a) is instantiated by rule CALL(a) to derive another caller fact $A(i2, i1)$, as visualized by arrow k in Fig. 4.2a.*

4.0.3 A Loophole in FlowDroid

A comparison between CLIPPER and FLOWDROID reveals a loophole in the latter. Although based on a flow-insensitive semantics, the goal of FLOWDROID is to implement a flow-sensitive analysis – a forward taint analysis to be specific – with an on-demand backward alias analysis piggybacked on it. Both forward/backware analyses are implemented in a framework for solving interprocedural finite distributive subset (*IFDS*) problems [22]. One shortcoming of this design is that the alias analysis is essentially intraprocedural-only because *backward* interprocedural data flows from callers to callees are actually propagated by the *forward* taint analysis, unlike CLIPPER where interprocedural data flows can be propagated in both directions – either forward through call arguments (by rule CALL(e)) or backward through returned value (by rule CALL(f)). Our next example in Fig. 4.5 illustrates CLIPPER has the ability to handle a case missed by FLOWDROID.

In this example, a *backward* data flow in `bar()` needs to be triggered by the alias analysis, as indicated by arrow *c'* in Fig. 4.5b. However, because the alias analysis of FLOWDROID triggers *forward* taint analysis in `bar()` only, further analysis of `bar()` is omitted and the leak at line 4 is missed, leading to a false negative. In contrast, rule CALL(f) of CLIPPER will trigger the analysis of `bar()` by generating the magic fact $A(ret.f.h, ret.f.h)$. The analysis of `bar()` reveals the alias fact $A(ret.f.h, ret.g.h)$ (along the path $c' \rightarrow d \rightarrow e \rightarrow f \rightarrow g'$) within `bar()` and thus the alias fact $A(w, x.g.h)$ at the caller which further exposes the leak at arrow h.

$$\frac{A(i2, i2) \quad CallRet(30, i2)}{A(ret_{get}, ret_{get})}\ (30)$$

(a) Query propagation at the call to `Vector.get()` at line 30.

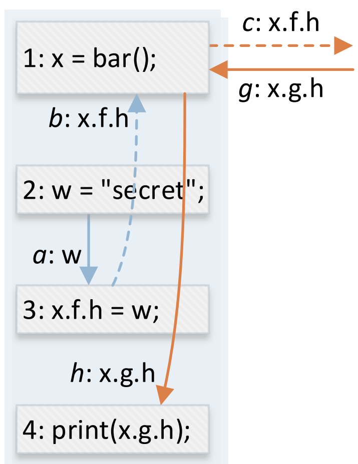
$$\frac{\dfrac{A(ret_{get}, ret_{get}) \quad OLoad(8, ret_{get}, a, [i])}{A(ret_{get}, a[i])}\ (8) \qquad OLoad(7, a, this_{get}, arr)}{A(ret_{get}, this_{get}.arr[i])}\ (7)$$

(b) Intraprocedural derivation within `Vector.get()`.

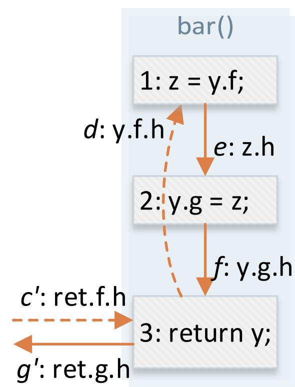
$$\frac{A(i2, i2) \quad CallRet(30, i2) \quad CallArg(30, 0, v\_main) \quad Param(get, 0, this_{get}) \quad A(ret_{get}, this_{get}.arr[i])}{A(i2, v\_main.arr[i])}\ (30)$$

(c) Instantiating the summary of `Vector.get()` at the call site at line 30.

Fig. 4.4.: Derivation examples. The numbers in parentheses are line numbers of corresponding statements.



(a) The derivation for facts of the form $A(w, \_)$.



(b) The derivation for facts of the form $A(ret.f.h, \_)$

Fig. 4.5.: Example illustrating the loophole in FlowDroid. The forward data flows denoted by solid arrows are implemented by the taint analysis while the backward data flows denoted by dashed arrows are implemented by the on-demand alias analysis. The data flows missed by FlowDroid are marked in red.

The root cause of the problem is the inconsistency between the flow-sensitivity of the analysis and the underlying semantics, i.e., FlowDroid tries to implement a flow-sensitive analysis over a flow-insensitive semantics (Section 3.0.4). In Chapter 6 we propose an example application demonstrating a sound way to integrate the flow-insensitive Clipper into a flow-sensitive analysis.

4.0.4 Static Fields and Jumping

Static fields represent *global* variables. The original description of FlowDroid handles static field as local variables [17] and load/store on static fields are handled in a similar way as assigment on *local* variables given the following generalized notation of access path:

$$x.\delta, \mathbb{f}.\delta \in AP = (Var \cup SField) \times \Delta$$

where the new form of access path $\mathbb{f}.\delta$ denotes those rooted at static field $\mathbb{f}$. Then the semantic functions with respect to load/store on static fields amount to assignment on global variables:

$$[\![l\colon x = \mathbb{f}]\!]^{\mathbb{A}}(A) := A \cup \{\mathbb{f}.\delta \mid x.\delta \in A\} \cup \{x.\delta \mid \mathbb{f}.\delta \in A\}$$
$$[\![l\colon \mathbb{f} = x]\!]^{\mathbb{A}}(A) := A \cup \{\mathbb{f}.\delta \mid x.\delta \in A\} \cup \{x.\delta \mid \mathbb{f}.\delta \in A\}$$

Since static fields are globally visible in all methods, the functions $A\downarrow$ and $A\uparrow$ are extended to:

$$A\downarrow = \{h_i.\delta \mid y_i.\delta \in A\} \cup \{ret_p.\delta \mid x.\delta \in A\} \cup \{\mathbb{f}.\delta \mid \mathbb{f}.\delta \in A\}$$
$$A\uparrow = \{y_i.\delta \mid h_i.\delta \in A\} \cup \{x.\delta \mid ret_p.\delta \in A\} \cup \{\mathbb{f}.\delta \mid \mathbb{f}.\delta \in A\}$$

which increase the analysis complexity – more rules to encode $[\![l\colon x{=}p(\overline{y})]\!]^{\mathbb{A}}$ and more facts derived.

Observation Different from local variables which only exist in the method where they are defined, static fields represent global variables which are valid in all methods, as are the access paths rooted at static fields. This causes a "jumping" effect when analyzing load/store on static fields – a load/store on static field $\mathbb{f}$ may interfere with any other load/store on $\mathbb{f}$. Thus the task of computing the alias class $[x.\delta]$ of a local query (local-variable-rooted access path like $x.\delta$) can be delegated to computing the alias class $[\mathbb{f}.\delta]$ of a global query (static-field-rooted access path like $\mathbb{f}.\delta$) by generating magic facts of the form $A(\mathbb{f}.\delta, \mathbb{f}.\delta)$ indicating "the problem of determining alias class of $\mathbb{f}.\delta$ arises", as shown by Rules SLOAD(a) and SSTORE(a) below.

$$[\![l\colon x = \mathbb{f}]\!]^{\mathbb{A}} \Rightarrow \begin{cases} A(\mathbb{f}.\delta, \mathbb{f}.\delta) \text{ :- } A(\alpha, x.\delta), SLoad(l, x, \mathbb{f}). & \text{(a)} \\ A(\mathbb{f}.\delta, x.\delta) \text{ :- } A(\mathbb{f}.\delta, \mathbb{f}.\delta), SLoad(l, x, \mathbb{f}). & \text{(b)} \end{cases} \quad \text{(SLOAD)}$$

$$[\![l\colon \mathbb{f} = x]\!]^{\mathbb{A}} \Rightarrow \begin{cases} A(\mathbb{f}.\delta, \mathbb{f}.\delta) \text{ :- } A(\alpha, x.\delta), SStore(l, \mathbb{f}, x). & \text{(a)} \\ A(\mathbb{f}.\delta, x.\delta) \text{ :- } A(\mathbb{f}.\delta, \mathbb{f}.\delta), SStore(l, \mathbb{f}, x). & \text{(b)} \end{cases} \quad \text{(SSTORE)}$$

These magic facts trigger further analysis at interfering load/store on $\mathbb{f}$ (hence "jumping"), as shown by Rules SLOAD(b) and SSTORE(b).

Similarly, a magic fact $A(\mathbb{f}.\delta, \mathbb{f}.\delta)$ holds at all call sites and hence queries for callee summaries of the form $A(\mathbb{f}.\delta, \beta)$, which should be instantiated at all call sites as shown below.

$$l\colon x{=}p(\overline{y}) \Rightarrow$$

$$\begin{cases} \qquad\qquad \dots & (a-f) \\ A(\mathbb{f}.\delta, y_i.\delta') \text{ :- } A(\mathbb{f}.\delta, h_i.\delta'), Param(p, i, h_i), CallArg(l, i, y_i). & (g) \\ A(\mathbb{f}.\delta, x.\delta') \text{ :- } A(\mathbb{f}.\delta, ret_p.\delta'), CallRet(l, x). & (h) \end{cases} \quad \text{(CALL)}$$

To skip expensive local propagation of alias facts for scalability, a similar approach can be generalized to object fields with the notation of access path further generalized below

$$x.\delta, \mathbb{f}.\delta, f.\delta \in AP = (Var \cup SField \cup OField) \times \Delta$$

Given a relation *Jump* specifying the set of object fields to be handled like static fields, the rules corresponding to load/store of object field can be modified as follows:

$$l\text{: } x{=}y.f \Rightarrow$$
$$\begin{cases} A(\alpha, y.f.\delta) \text{ :- } A(\alpha, x.\delta), OLoad(l, x, y, f), \overline{\boldsymbol{Jump}(f)}. & \text{(a)} \\ A(\alpha, x.\delta') \text{ :- } A(\alpha, y.\delta), OLoad(l, x, y, f), \delta'{\in}\delta_{\overline{f}}, \overline{\boldsymbol{Jump}(f)}. & \text{(b)} \\ A(f.\delta, f.\delta) \text{ :- } A(\alpha, x.\delta), OLoad(l, x, y, f), \boldsymbol{Jump}(f). & \text{(c)} \\ A(f.\delta, x.\delta) \text{ :- } A(f.\delta, f.\delta), OLoad(l, x, y, f), \boldsymbol{Jump}(f). & \text{(d)} \end{cases} \qquad \text{(OLOAD2)}$$

$$l\text{: } y.f{=}x \Rightarrow$$
$$\begin{cases} A(\alpha, y.f.\delta) \text{ :- } A(\alpha, x.\delta), OStore(l, y, f, x), \overline{\boldsymbol{Jump}(f)}. & \text{(a)} \\ A(\alpha, x.\delta') \text{ :- } A(\alpha, y.\delta), OStore(l, y, f, x), \delta'{\in}\delta_{\overline{f}}, \overline{\boldsymbol{Jump}(f)}. & \text{(b)} \\ A(f.\delta, f.\delta) \text{ :- } A(\alpha, x.\delta), OStore(l, y, f, x), \boldsymbol{Jump}(f). & \text{(c)} \\ A(f.\delta, x.\delta) \text{ :- } A(f.\delta, f.\delta), OStore(l, y, f, x), \boldsymbol{Jump}(f). & \text{(d)} \end{cases} \qquad \text{(OSTORE2)}$$

For fields excluded from the relation *Jump*, rules OLOAD2(a-b) and OSTORE2(a-b) handle their loads and store as usual. For fields included in *Jump*, rules OLOAD2(c-d) and OSTORE2(c-d) handle their loads and stores like static fields.

Example 5 *Given a set of jumping fields* $Jump = \{names\}$ *and the access path* $v\_names.arr[i]$ *at the load from the names field at line 23 (arrow d in Fig. 4.7), it "jumps" to*

1. *the store at line 18 as derived in Fig. 4.6a and visualized by arrow e in Fig. 4.7;*

2. *the load at line 20 as derived in Fig. 4.6b and visualized by arrow g in Fig. 4.7.*

A field-rooted access path like $names.arr[i]$ essentially over-approximates the set of variable-rooted ones like $x.\delta.names.arr[i]$, had the access path $v\_names.arr[i]$ been

$$\dfrac{\dfrac{A(name, v\_names.arr[i]) \quad OLoad(23, v\_names, this_{fetch}, names) \quad Jump(names)}{A(names.arr[i], names.arr[i])}\ (23) \qquad OStore(18, this_{AddrBook}, names, v\_names)}{A(names.arr[i], v\_names.arr[i])}\ (18)$$

(a) Derivation of jumping to "18: $this.names = v\_names$".

$$\dfrac{\dfrac{A(name, v\_names.arr[i]) \quad OLoad(23, v\_names, this_{fetch}, names) \quad Jump(names)}{A(names.arr[i], names.arr[i])}\ (23) \qquad OLoad(20, v\_names, this_{update}, names)}{A(names.arr[i], v\_names.arr[i])}\ (20)$$

(b) Derivation of jumping to "20: $v\_names = this.names$".

Fig. 4.6.: Jumping example. The numbers in parentheses are line numbers of corresponding statements.

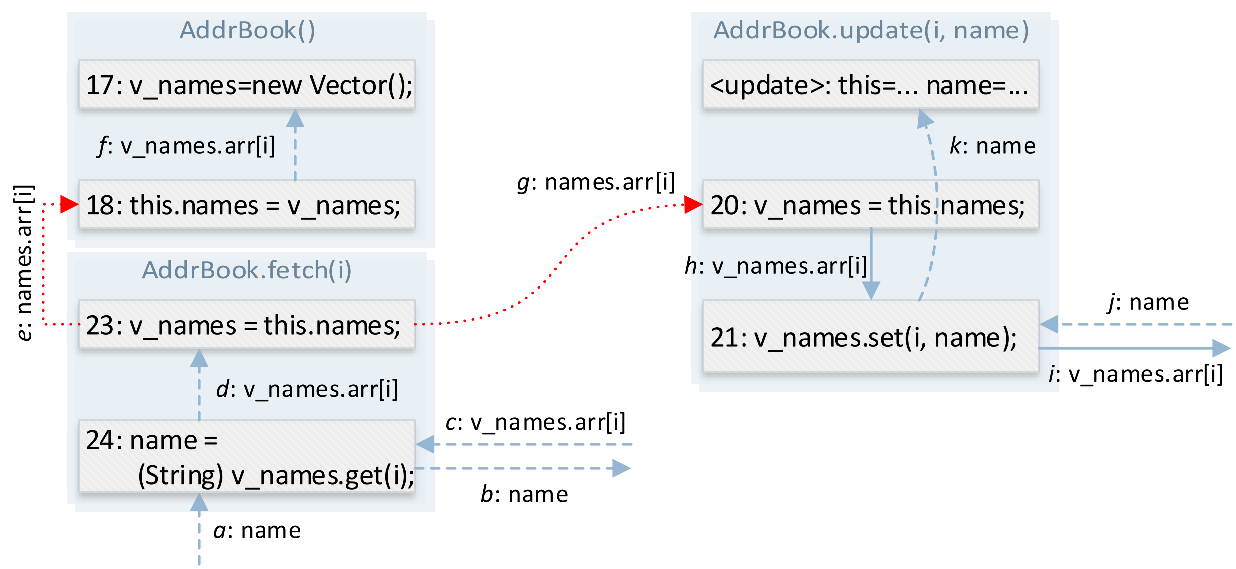


Fig. 4.7.: Visualizing jumping as data flows. The dotted red arrows represent jumping flows.

propagated locally at line 23. This may introduce spurious facts inferred but will not affect the soundness of the analysis.

In our experiments, only instance fields defined outside library packages are jumpable and we identify such library packages with their names, including "`java.lang.*`" and "`java.util.*`".

4.1 Access Path Abstraction

In real-world object-oriented programs, recursive data structures and abstract data types are two most commonly used programming constructs. An *Abstract Data Type* (*ADT*) is an interface specification of classes, i.e., a description of the data they represent and the permissible operations (i.e., methods) on these data [23]. For example, the `List` interface defined in Fig. 4.8, formed by its permissible methods `get()` and `set()`, is an example of an *ADT*.

```
interface List {
  Object get(int i);
  void set(int i, Object x);
}
```

Fig. 4.8.: Example code for `List` interface.

Both `Vector` and `LinkedList` classes implement these methods and thus the `List` interface (Fig. 4.9). Particularly, the linked list implemented by the `LinkedList` class is a commonly used recursive data structure.

These two programming constructs pose challenges to scalability of access-path-based heap model. An access path abstraction mechanism to automatically overcome these challenges is introduced in the next two sections.

```
class Vector implements List {...}
class Node {Node next; Object e;}
class LinkedList implements List {
  Object get(int i) {
    Node n = this.head;
    for(; i>0; i--) {n = n.next;}
    return n.e;}
  void set(int i, Object x) {
    Node n = this.head;
    for(; i>0; i--) {n = n.next;}
    n.e = x;}
}
class AddrBook {
  List names;
  AddrBook(boolean useVector) {
    if(useVector) {this.names = new Vector();}
    else {this.names = new LinkedList();}}
}
```

Fig. 4.9.: Example code for illustrating access path abstraction.

,

4.1.1 Cyclic Pattern Reduction

Recursive data structures are usually accessed with recursive program constructs such as loops and recursive method calls. In `LinkedList.set()` for example, accesses to list elements start from the head node (line 9) and advance from one node to its successor in each iteration of the loop (line 10). Such iterations cause access paths to grow indefinitely (arrow d_k in Fig. 4.10), generating an unbounded set of method summaries $\{A(x, \alpha) \mid \alpha \in this.head(.next)^*.e\}$, where the access path pattern $this.head(.next)^*.e$ represents the infinite set of access paths $\{this.head.e$, $this.head.next.e$, $this.head.next.next.e$, ...$\}$.

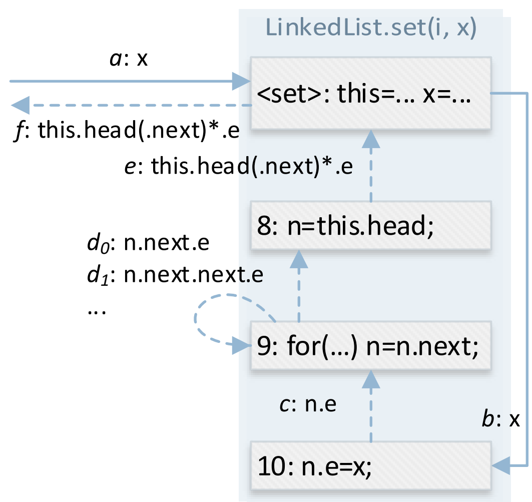


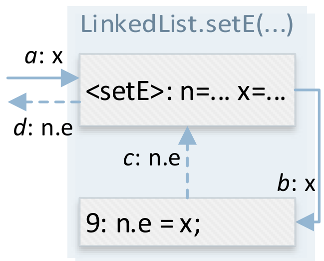
Fig. 4.10.: Unbounded access paths generated by a loop iteratively.

Accesses to recursive data structures can also be implemented with recursive method calls. In an alternative linked list implementation in Fig. 4.11, accesses to list elements are performed by recursively invoking (line 8) an internal setter method `setE()` which advance from one node to its successor at line 7. By applying the generated method summary repeatedly at the recursive call site, the generated access paths may grow indefinitely, as shown in Fig. 4.12. The flow path through the `if` branch generates the initial summary $A(x, n.e)$ (Fig. 4.12a). Instantiating this summary at the recursive call in the `else` branch generates the second summary

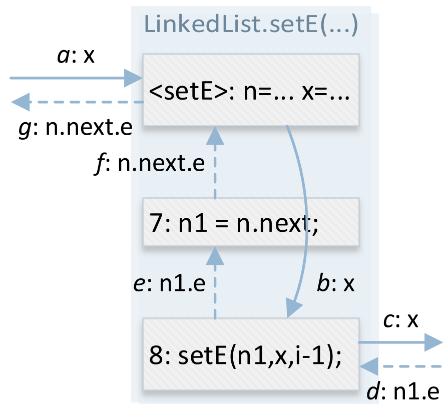
```
class LinkedList {
  void set(int i, Object x) {
    Node n = this.head;
    setE(n,x,i);}
  void setE(
    Node n, Object x, int i) {
    if(i>0) {Node n1 = n.next;
             setE(n1,x,i-1);}
    else if(i==0) {n.e = x;}}
}
```

Fig. 4.11.: Traversing recursive data structure via recursive method calls.

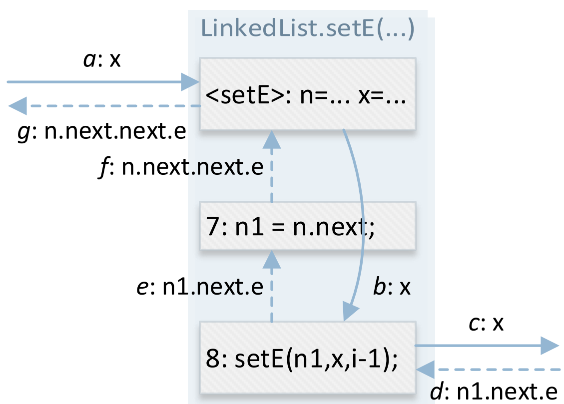
,



(a) Generating method summary $(x, n.e)$ from the `if` branch.



(b) Applying the summary $(x, n.e)$ to the recursive call within the `else` branch to generate new summary $(x, n.next.e)$.



(c) Applying the summary $(x, n.next.e)$ to the recursive call within the `else` branch to generate new summary $(x, n.next.next.e)$.

Fig. 4.12.: Unbounded access paths generated by recursive method calls.

$A(x, n.next.e)$ (Fig. 4.12b). Instantiating the second summary generates the third summary $A(x, n.next.next.e)$ (Fig. 4.12c). Repeating the above process generates the same unbounded set of method summaries $\{A(x, \alpha) \mid \alpha \in this.head(.next)^*.e\}$.

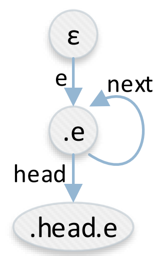


Fig. 4.13.: Set of field paths encoded as a *DFA*.

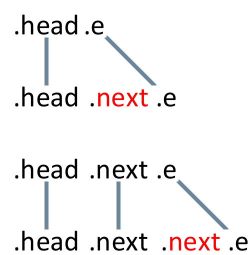


Fig. 4.14.: String matching example.

$$\delta_f := \begin{cases} \delta_2 & \text{if } f.\delta = \delta_1.\delta_2 \wedge \delta_1 \in CYC \\ f.\delta & \text{otherwise} \end{cases}$$

$$\delta_{\overline{f}} := \begin{cases} \{\delta'\} & \text{if } \delta = f.\delta' \\ \{\delta'.\delta \mid f.\delta' \in CYC\} & \text{if } \exists \delta' : f.\delta' \in CYC \\ \emptyset & \text{otherwise} \end{cases}$$

Fig. 4.15.: Modified δ_f and $\delta_{\overline{f}}$ to support cycle reduction.

Denoted as a regular expression, the set of field paths $head.(next.)^*e$ can be encoded as the *deterministic finite automaton* (*DFA*) [24] in Fig. 4.13, where each state is labeled with one of the field paths it represents.

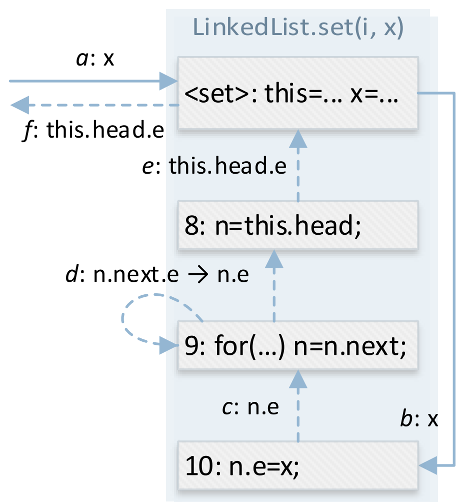


Fig. 4.16.: Bounded access paths abstracted via cycle reduction.

The cyclic pattern "*next*" in the *DFA* can be extracted from the aforementioned set of method summaries $\{A(x,\alpha) \mid \alpha \in this.head(.next)^{*}.e\}$, which implies the alias relation among the access paths "$this.head(.next)^{*}.e$". Then detecting the cyclic pattern can be generalized to an *approximate string matching* problem (also known as *edit distance* problem in [25]), where each field path is denoted by a string and each field in the field path is denoted by a character in the string. As shown in Fig. 4.14, the *inserted* part "*next*" between matching parts constitutes the cyclic pattern. This inference method is both general and effective. It doesn't require extra conditions on the program structure such as reducibility or extra analysis to extract this structure information with interval analysis. In our experiments, it successfully inferred all cyclic patterns in the Java Class Library and benchmark programs.

Given a set of cyclic patterns $CYC \subseteq \Delta^{\#}$, the original definition of δ_f and $\delta_{\overline{f}}$ in Section 3.0.4 can be modified to encode the transition relation of the corresponding *DFA* (Fig. 4.15), i.e., $\delta \xrightarrow{f} \delta_f$ and $\forall \delta' \in \delta_{\overline{f}} : \delta' \xrightarrow{f} \delta$. Such *DFA*-based field path encoding generates only bounded access paths by reducing cycles automatically, as shown in Fig. 4.16 where $CYC = \{next\}$.

4.1.2 Abstract Data Type Based Reduction

*ADT*s pose another challenge to the scalability of access-path-based analysis because the same conceptual reference relation can be encoded by different field paths of different implementations. In Fig. 4.9, for example, the collection of names referenced via `AddrBook.names` could be implemented with either `Vector` or `List`. Thus the conceptual reference relation between the `AddrBook` object and its name objects could be encoded with either "$names.addr[i]$" or "$names.head.e$", as shown in Fig. 4.17. The situation becomes worse in real-world programs due to multiple implementations of the same interface type and nesting of collection types like `Map<String,List>`.

The reference relation between `List` objects and their elements can be modeled with a pseudo field ***elem*** (Pseudo fields are denoted in bold face,) which can be implemented as field paths $arr[i]$ (by `Vector`) or $head.e$ (by `LinkedList`). These field paths can be automatically extracted from summaries of implementations of the setter method `List.set(i,x)`, e.g. $A(this.arr[i], x)$ in Fig. 3.5b and $A(x, this.head.e)$ in Fig. 4.16.

Similarly for the `java.util.Map` interface, different field paths implementing the conceptual reference relation between map objects and their keys and values (modeled with pseudo fields $\boldsymbol{key}$ and $\boldsymbol{value}$ respectively) can be inferred from implementations of the setter method `Map.put(key,value)`. The set of abstract fields ($OField^{\#}$) is defined as the union of these pseudo fields and the real fields ($Field$) defined above, i.e., $f^{\#}, g^{\#} \in OField^{\#} = OField \cup \{\boldsymbol{elem}, \boldsymbol{key}, \boldsymbol{value}\}$. Then an abstract field path is a sequence of such abstract fields, i.e., $\delta^{\#} \in \Delta^{\#} = OField^{\#*}$.

The pseudo fields and their alternative implementing field paths can be modeled by production rules (e.g. $\boldsymbol{elem} \rightarrow arr[i] \mid head.e$,) which can be encoded as pairs in a relation called ADT, i.e., $(f^{\#}, \delta^{\#}) \in ADT \subseteq OField^{\#} \times \Delta^{\#}$ if $f^{\#} \rightarrow \delta^{\#}$. In our example, the inferred ADT relation is $\{(\boldsymbol{elem}, arr[i]), (\boldsymbol{elem}, head.e)\}$.

The definition of $\delta^{\#}_{f}$ and $\delta^{\#}_{f}$ can be further modified to support field path reduction with respect to the production rules in ADT:

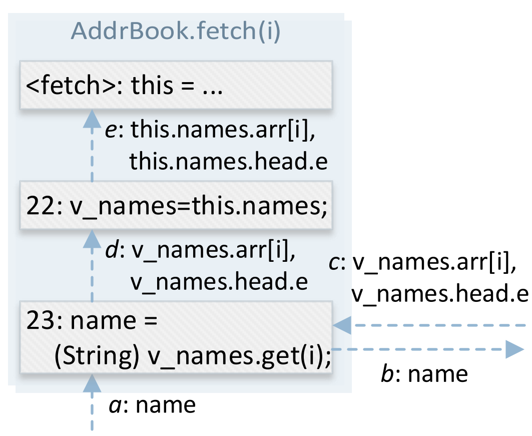


Fig. 4.17.: Multiple access paths caused by ADT.

Fig. 4.19 shows the effects of applying ADT-based reduction on our example analysis.

$$\delta_f^{\#} := \begin{cases} \delta_2^{\#} & \text{if } f.\delta^{\#} = \delta_1^{\#}.\delta_2^{\#} \wedge \delta_1^{\#} \in CYC \\ f^{\#}.\delta_2^{\#} & \text{if } f.\delta^{\#} = \delta_1^{\#}.\delta_2^{\#} \wedge (f^{\#}, \delta_1^{\#}) \in ADT \\ f.\delta^{\#} & \text{otherwise} \end{cases}$$

$$\delta_{\bar{f}}^{\#} := \begin{cases} \{\delta^{\#\prime}\} & \text{if } \delta^{\#} = f.\delta^{\#\prime} \\ \{\delta_1^{\#}.\delta_2^{\#} \mid (f^{\#}, f.\delta_1^{\#}) \in ADT\} & \text{if } \begin{cases} \delta^{\#} = f^{\#}.\delta_2^{\#} \text{ and} \\ \exists \delta_1^{\#} : (f^{\#}, f.\delta_1^{\#}) \in ADT \end{cases} \\ \{\delta^{\#\prime}.\delta^{\#} \mid f.\delta^{\#\prime} \in CYC\} & \text{if } \exists \delta^{\#\prime} : f.\delta^{\#\prime} \in CYC \\ \emptyset & \text{otherwise} \end{cases}$$

Fig. 4.18.: Modified $\delta_f^{\#}$ and $\delta_{\bar{f}}^{\#}$ to support ADT-based reduction.

Fig. 4.19.: Examples of ADT-based field path reduction.

4.1.3 Soundness of Access Path Abstraction

Given the definitions of abstract fields and field paths above, we further define:

$$\text{abstract access path } \alpha^{\#}, \beta^{\#}, \langle x, \delta^{\#} \rangle \in AP^{\#} = Var \times \Delta^{\#}$$
$$\text{abstract alias class } A^{\#} \in AClass^{\#} = 2^{AP^{\#}}$$

The semantic function $[\![\cdot]\!]^{\#}$ for the abstract denotational semantics of our simple language can be obtained by replacing δ_f and $\delta_{\overline{f}}$ with $\delta^{\#}_f$ and $\delta^{\#}_{\overline{f}}$ defined in Fig. 4.18. An extraction function $\eta : AP \to AP^{\#}$ to "extract" the abstract access path representing a given concrete one can be defined by iteratively applying function $\delta^{\#}_{f_i}$ on each field f_i of the concrete access path, i.e $\eta(x.f_1.f_2...f_{k-1}.f_k) = x.((((\epsilon_{f_k})_{f_{k-1}})_{...})_{f_2})_{f_1}$. Then a Galois connection $(AClass, \boldsymbol{\alpha}, \boldsymbol{\gamma}, AClass^{\#})$ can be defined as in [20]:

$$\boldsymbol{\alpha}(A) = \{\eta(\alpha) \mid \alpha \in A\} \text{ and } \boldsymbol{\gamma}(A^{\#}) = \{\alpha \mid \eta(\alpha) \in A^{\#}\}$$

Theorem 4.1.1 $[\![\cdot]\!]^{\#}$ *is a correct upper approximation of* $[\![\cdot]\!]^{\mathbb{A}}$.

Proof A case-by-case analysis of the semantic functions of each statement to verify that
$\forall stmt$: $\forall A^{\#}$: $\boldsymbol{\alpha}([\![stmt]\!]^{\mathbb{A}}(\boldsymbol{\gamma}(A^{\#}))) \sqsubseteq [\![stmt]\!]^{\#}(A^{\#})$ ■

Next we give two applications of CLIPPER.

5. IMPLICIT CONTROL FLOW ANALYSIS

Event-driven frameworks such as Android[1] allow clients to register callbacks for various events. In Fig. 1.4 for example, the framework defines the "paused" event for activities. Listeners may be registered by clients for such events via the framework method `App.register()`. Clients may customize their own event handling methods by implementing the interface `ICallback.onPaused()`. In this example, the client method `MyApp.onCreate()` registers a customized listener object of type `MyCallback`, which is invoked by the framework at line 14.

For scalability purposes, many static analyses such as FlowDroid abstract the framework part of the program with a simplified model, and the causal relation between the registration and the callback is modeled as implicit control flow. In FlowDroid, for example, such a causal relation is specified *manually* as the registration/-callback pair: `App.register()` $\rightarrow$ `ICallback.onPaused()`.

We propose to infer this causal relation *automatically* with Clipper, as demonstrated in Fig. 5.1. A query with respect to the callee's receiver is issued at each *callback edge* (e.g. arrow a in Fig. 5.1) within the call-graph where

- The call site is within the framework part, e.g. line 14;
- The callee is a client-defined method, e.g. `MyCallback.onPaused()`.

Then the alias access paths are propagated until reaching a *registration edge* (e.g. arrow k) where

- The call site is within the client part, e.g. line 24;
- The callee is a framework method, e.g. `App.register()`.

[1]https://www.android.com/

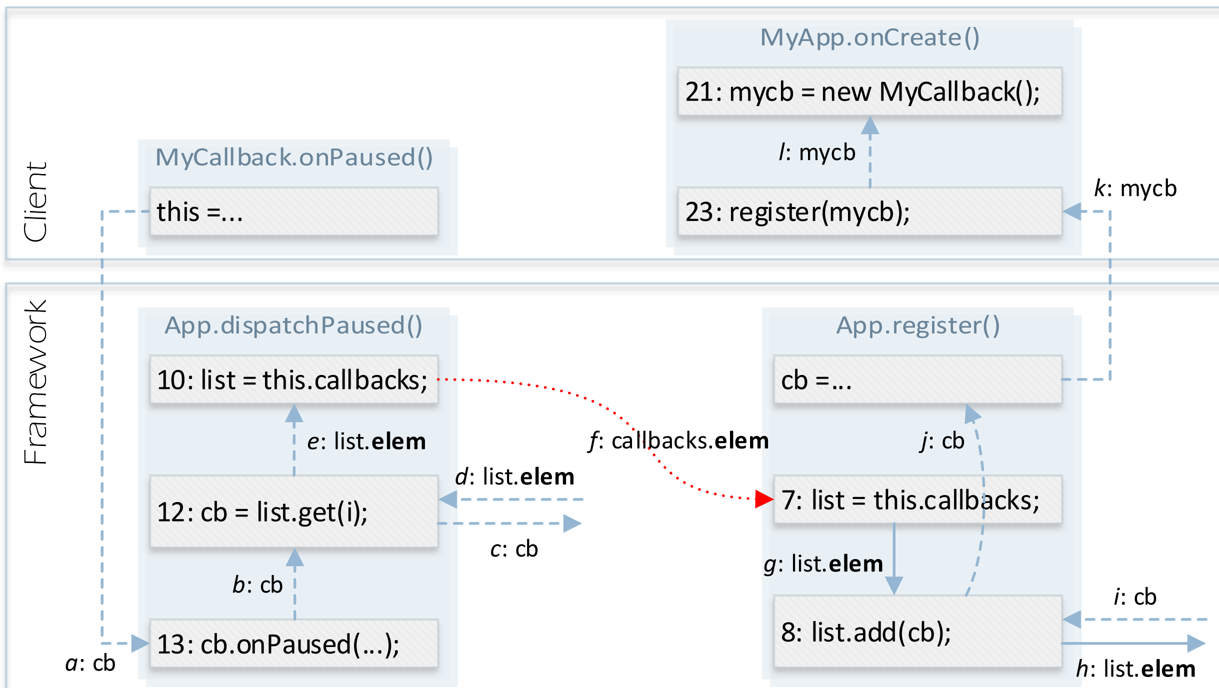


Fig. 5.1.: Inference of implicit control flows within the program of Fig. 1.4. The solid/dashed/red arrows have the same meaning as before.

Hence the registration/callback pair `App.register()` $\rightarrow$ `ICallback.onPaused()` is exposed.

For each given client part, the inference analysis implemented will automatically generate a set of registration/callback pairs potentially used by the client. The automatically generated set of pairs can be used as drop-in replacement of the manually specified one in FlowDroid. Our experimental results show FlowDroid based on automatically generated registration/callback pairs achieves the same precision as the one based on manually specified ones.

5.1 Limitation of EdgeMiner

The implicit control flow analysis is initially inspired by EdgeMiner [26]. Same as the Clipper based implicit control flow analysis in this chapter, EdgeMiner is used to infer a set of registration-callback pairs. Different from the Clipper based approach, EdgeMiner applies a field-insensitive heap model, which is one major drawback of EdgeMiner's design. Consider the example program in Fig. 5.2 where a callback object is register indirectly via a `Map` object. Hence the callback object is stored to the `Map` object via the invocation of method `Map.put()` at line 16 before the `Map` object is stored to the field `callbacks` via the invocation of method `register()` at line 17 as shown in Fig. 5.3.

Due to the field-insensitive model applied by EdgeMiner, it will incorrectly infer the registration site as the invocation of method `Map.put()` at line 16 rather than the invocation of method `register()` at line 17.

The heap model applied by Clipper is access-path based and hence field-sensitive, which can correctly handle a senario like this.

```
class Session {
  Map callbacks;
  void register(Map cbs) {
    this.callbacks = cbs;
  }
  void flush() {
    Map cbs = this.callbacks;
    Listener cb = cbs.get("flush");
    cb.onFlush();
  }
}
void main() {
  Session session = openSession();
  Map cbs = new HashMap();
  Listener cb = new FlushEventListener();
  cbs.put("flush", cb);
  session.register(cbs);
  session.beginTransaction();
  ...
}
```

Fig. 5.2.: Example code where a field-sensitive heap model is needed for implicit control flow analysis.

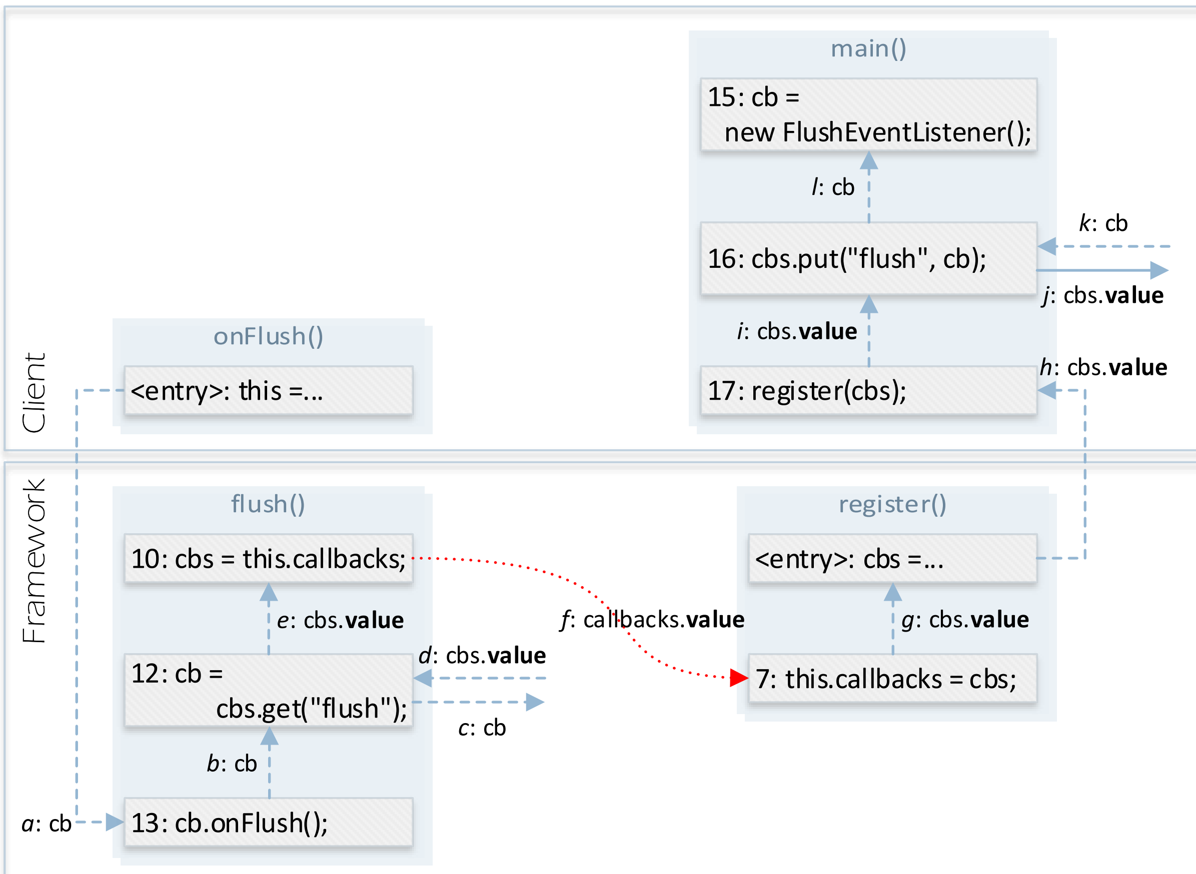


Fig. 5.3.: Example code where a field-sensitive heap model is needed for implicit control flow analysis.

6. DYNASENS – A DEMAND-DRIVEN POINTS-TO ANALYSIS

As demonstrated in Section 4.0.3, it is unsound to add flow-sensitivity to an analysis based on an inherently flow-insensitive semantics. However, if focusing on a program's *effect* as sets of alias classes determined by it, as explained in Section 3.0.4, a heap analysis like CLIPPER can be used as a *slicing analysis*: given a query (access path) α as the *slicing criterion*, CLIPPER generates a *slice* – a subset of the program's elements preserving the program's effect – the alias class of α. The slice can be used to refine another flow-sensitive analysis – focusing the latter on the program elements within the slice – thus forming an iterative analysis as a whole [27].

In this chapter, we present DYNASENS – a demand-driven approach to automatically refine a (flow-sensitive) points-to analysis by adjusting its context-sensitivity with CLIPPER.

6.0.1 Overview of Parametric Points-to Analysis

Within a points-to graph generated by a points-to analysis, objects are modeled with global names such as Vec_{name} and Vec_{Int} denoting the vectors allocated at line 17 and 27, respectively, and Arr denoting the array allocated at line 4. As an example, Fig. 3.1 shows part of the data flow generated by a context-insensitive points-to analysis of the example program in Fig. 1.1. Due to absence of context, the flows of Vec_{name} and Vec_{Int} merge along paths $a \to c$ and $b \to c$ respectively, where their own arrays, denoted by the common name Arr, are modeled as being stored to their `arr` field, as shown in the generated points-to graph in the bottom-right corner of Fig. 3.1. Similarly, due to confluence of these two vectors along paths $f \to g \to l$ and $h \to j \to l$, and confluence of the string *name* and integer 3 along paths $e \to g \to k$

and $i \to j \to k$, both *name* and 3 are modeled as being stored to *Arr* at line 11, as shown in the generated points-to graph.

Finer points-to graphs can be generated by augmenting data flows with contexts. Smaragdakis et al. developed a parametric points-to analysis which allow manual specification of different context sensitivities on different program elements [28], so that a subset of the program elements are analyzed context-sensitively while the rest are analyzed context-insensitively. Such configurability comes from a parametric redesign of the points-to analysis rules and the introduction of two new input relations to configure these rules. The input relations used to configure the analysis are shown below:

- $\boldsymbol{ObjectToRefine}(lalloc)$: a set of allocation sites ($lalloc$) where context sensitivity is enabled.
- $\boldsymbol{SiteToRefine}(lcall, p)$: a set of call edges from call sites ($lcall$) to callees (p) where context sensitivity is enabled.

The parametric rule handling object allocation is:

$$VarPointsTo(x, ctx, l, hctx) \leftarrow Alloc(l, x).$$

$$\text{where } \boldsymbol{hctx} = \begin{cases} * & \text{if } l \notin ObjectToRefine \\ \boldsymbol{RecordRefined}(l, ctx) & \text{if } l \in ObjectToRefine \end{cases}$$

where an allocation "l: $x =$ new t" generates an abstract object whose quantifying context $hctx$ may be context insensitive (denoted by "$*$"), or a refined context, computed by the function $RecordRefined$ depending on the configuring input relation $ObjectToRefine$.

The parametric rule handling call graph building is:

$$CallGraph(l_{call}, callerCtx, p, calleeCtx) \leftarrow$$
$$Call(l_{call}, y_0, p), PointsTo(y_0, callerCtx, l_{base}, baseHCtx).$$

$$\text{where } \boldsymbol{calleeCtx} = \begin{cases} * \quad \text{if } \langle l_{call}, p\rangle \notin SiteToRefine \\ \boldsymbol{MergeRefined}(l_{base}, baseHCtx, l_{call}, callerCtx) \\ \qquad\qquad\qquad\qquad \text{if } \langle l_{call}, p\rangle \in SiteToRefine \end{cases}$$

where an invocation "l_{call}: $x{=}p(\overline{y})$" on the receiver object represented by $\langle l_{base}, baseHCtx\rangle$ generates a call edge to callee p, whose quantifying context $calleeCtx$ may be either context-insensitive (denoted by $*$), or refined and computed by the function $MergeRefined$, depending on the configuring input relation $SiteToRefine$.

Smaragdakis et al. [28] uses heuristic rules to populate input relations $ObjectToRefine$ and $SiteToRefine$ to configure the analysis for better precision and scalability.

We propose to populate these input relations with elements from the slice generated by Clipper as a slicing analysis so that the configuration is guided by the demand from certain client analysis.

6.0.2 Populating Input Relations

The configuring input relations $ObjectToRefine(l)$ and $SiteToRefine(l, p)$ of the parametric points-to analysis can be populated from the analysis result of Clipper with the following rules, where $CallGraph$ is generated by a bootstrapping context-insensitive points-to analysis.

$$ObjectToRefine(l) \text{ :- } A(\alpha, x.\delta), Alloc(l, x).$$
$$SiteToRefine(l, p) \text{ :- } A(\alpha, y_i.\delta), CallArg(l, i, y_i), CallGraph(l, \_, p, \_).$$
$$SiteToRefine(l, p) \text{ :- } A(\alpha, x.\delta), CallRet(l, x), CallGraph(l, \_, p, \_).$$

In our example program, given the slicing criteria $i2$ encoded as a query to Clipper, the slice includes the following program elements from Fig. 4.4 (and 4.2) (labels are denoted by line numbers)

$$ObjectToRefine = \{4, 27, 28\}$$

$$SiteToRefine = \{(27, \texttt{Vector()}), (29, \texttt{Vector.set()}), (30, \texttt{Vector.get()})\}$$

Given the above configuration, the refined points-to analysis is shown in Fig. 6.1. Since the call site at line 27 is determined to be context-sensitive, the callee `Vector()` is specialized with the receiver Vec_{Int} as its context. So is the array Arr_{Int} allocated within it at line 4. Thus the context-sensitive flows b, c_2, and d_2 are separated from the context-insensitive ones a, c_1, and d_1. Similarly, since the call site at line 29 is determined to be context-sensitive, Thus the context-sensitive flows j, k_2, l_2, and m_2 are separated from the context-insensitive ones g, k_1, l_1, and m_1. As a result, the contents of two vectors Vec_{name} and Vec_{Int} are separated in the generated points-to graph in Fig. 6.2.

6.0.3 Experiment Setting

We give experimental results for the following points-to analyses, named according to the naming convention in [29]:

Insen The context-insensitive analysis.

2type+1H 2-type-sensitive analysis with a 1-type-sensitive heap [29], which is optimized for scalability.

1type1obj+1H 1-type-1-object-sensitive analysis with a 1-object-sensitive heap [29], which shows the best trade-off between scalability and precision among other type-sensitive analyses.

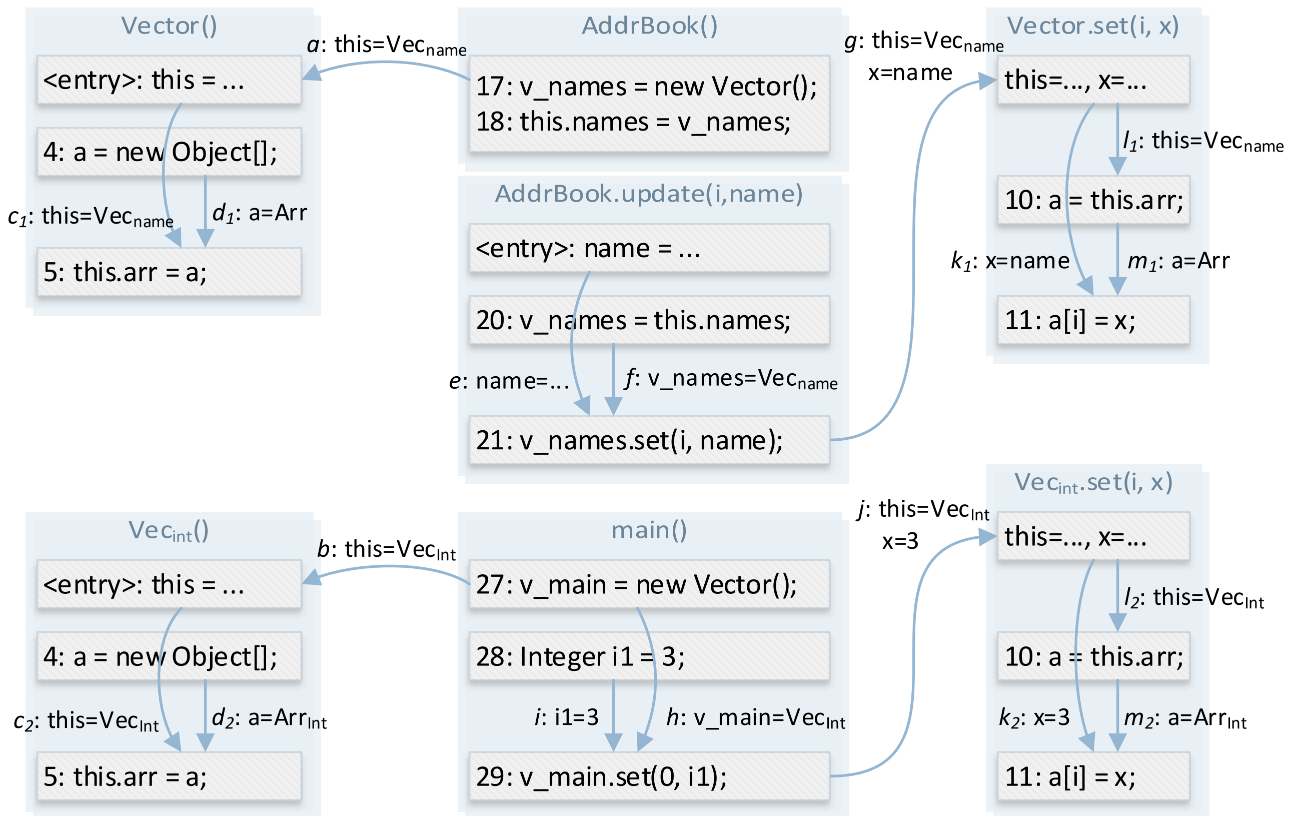


Fig. 6.1.: Part of the data flows of the refined points-to analysis on the example program in Fig. 1.1.

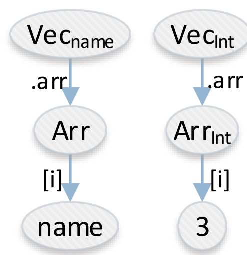


Fig. 6.2.: Part of points-to graph generated by the refined points-to analysis.

DynaSens The refined analysis which is selectively context-sensitive on the elements of the input relations populated by CLIPPER with respect to certain query. Furthermore, it is a parametric analysis which can be configured to yield any n-object-sensitive analysis with an (n-1)-object-sensitive heap (i.e., *(n)obj+(n-1)H*) [2]. Such configurability comes from the following context constructors for *(n)obj+(n-1)H*:

- $RecordRefined(l, ctx) = [ctx[1], ..., ctx[n-1]$
- $MergeRefined(l_{base}, baseHCtx, l_{call}, callerCtx) =$
$$[l_{base}, baseHCtx[1], ..., baseHCtx[n-1]]$$

where $ctx, callerCtx \in (Label \cup \{*\})^n$ and $baseHCtx \in (Label \cup \{*\})^{n-1}$ are sequences of allocation sites and $*$ (a unique constant value), of length n and $n-1$ respectively, and $ctx[i]$ (or $hctx[i]$) is the i-th element of ctx (or $hctx$), starting with index 1.

We evaluated DYNASENS on Xeon E5-2660 2.6GHz machines with 64GB of RAM with two clients: (1) downcast safety checking (Section 6.0.4) and (2) copy constant propagation (Section 6.0.5).

6.0.4 Downcast Safety Checking

The DaCapo benchmark suite [30], v.9.12-bach and v.2006-10-MR2, is used to test the downcast safety analysis. We analyzed most benchmarks (Table 6.1) used in previous work [29], ranging in size from 368K to 2324K bytecodes. The *jython* benchmark is excluded because it generates bytecode DynSamically during run-time. The benchmarks *luindex* and *lusearch* are combined into one benchmark *lucene* because they are actually two drivers of the same library *lucene*.

For each downcast site in the benchmark program, the client analysis tries to prove its safety by configuring the points-to analysis with the result of the slicing analysis carried out with respect to the downcast. The program elements from the slicing result are configured with 2obj+1H context-sensitivity. Table 6.2 shows the detailed

Table 6.1.: Benchmark characteristics. The *Classes* and *Methods* columns show the number of classes and methods in the benchmark. The *Bytecodes* column shows the size of the benchmark in Kbytes. The *Queries* column shows the number of downcast sites in the application classes of the benchmark.

Name	Classes	Methods	Bytecodes (Kbyte)	Queries
antlr	968	6161	368	127
lucene	1295	8583	485	147
bloat	1189	8979	530	1024
avrora	2306	12005	567	424
sunflow	1812	10610	602	156
chart	2129	18305	1048	551
xalan	2126	15576	893	901
pmd	2202	14852	916	1076
batik	3706	18614	1047	976
fop	5438	37496	2324	1487

results of our experiments, including the precision and cost metrics. The “reachables” rows show the number of reachable methods. The “call-graph (K)” rows show the number of thousands of nodes/edges within the context-sensitive call graph built. The “safe casts” rows show the number of safe downcasts proven by each points-to analysis. The greatest numbers are in **bold font**. The “points-to (K)” rows show the number of thousands of pairs of a context-sensitive local variable and an object within the points-to relation built. The “time” rows show the run-time of the (slicing and) points-to analyses in seconds. The shortest time is underlined.

Precision The precision metric is the number of casts that could be statically proven safe (the “safe casts” rows in Table 6.2). Thus higher numbers are better and the best result of each row is emphasized in bold font. As can be seen, DynaSens proves more casts safe than other analyses on most benchmarks (8 out of 10) and is very close to the most precise one on others.

Cost Cost is shown with three metrics: running time, the size of the *context-sensitive* call-graph, and size of points-to relation between *context-sensitive* local variables and objects. The cost of DynaSens is represented by the average value measured over all refined analyses for all cast queries. Different from running time which is an implementation-dependent measurement, the sizes of the context-sensitive call-graph and points-to relation are implementation-independent. The shortest running time for each benchmark is underlined in Table 6.2 for clarity.

As can be seen, DynaSens has the desirable feature of low cost: its running time for any benchmark is close to the context-insensitive one. This advantage is more salient on large benchmarks where DynaSens incurs the lowest cost among all context-sensitive analyses. This is because context sensitivity is enabled only on elements relevant to each query.

Table 6.2.: Results for downcast safety checking.

	Metrics	Insen	2type+1H	1type1obj+1H	DynaSens (2obj+1H)
antlr	reachables	3333	3293	3293	3333
	call-graph (K)	3/22	10/45	14/100	4/24
	points-to (K)	266	400	585	286
	safe casts	0/127	42/127	44/127	**65/127**
	time (sec)	19	23	34	1+19
lucene	reachables	4590	4498	4489	4590
	call-graph (K)	5/22	18/79	26/162	6/24
	points-to (K)	482	791	1413	559
	safe casts	0/147	85/147	**111/147**	104/147
	time (sec)	28	43	58	3+32
bloat	reachables	5537	5395	5388	5531
	call-graph (K)	6/41	27/192	40/308	12/85
	points-to (K)	1437	1992	3078	1721
	safe casts	0/1024	85/1024	114/1024	**220/1024**
	time (sec)	42	56	78	15+49
avrora	reachables	8560	8494	8493	8560
	call-graph (K)	9/37	39/140	53/225	11/42
	points-to (K)	1715	1758	2326	1719
	safe casts	0/424	99/424	100/424	**130/424**
	time (sec)	44	56	73	5+44
sunflow	reachables	5233	4997	4985	5232
	call-graph (K)	5/22	18/67	27/175	7/26
	points-to (K)	536	637	1410	624
	safe casts	0/156	83/156	85/156	**88/156**
	time (sec)	32	34	55	4+36
chart	reachables	8546	8327	8300	8546
	call-graph (K)	9/42	45/196	91/651	13/62
	points-to (K)	3207	3354	10311	3242
	safe casts	0/551	180/551	**270/551**	265/551
	time (sec)	75	98	183	6+80
xalan	reachables	9708	9114	9089	9708
	call-graph (K)	10/58	55/677	129/1729	15/111
	points-to (K)	2118	4403	16888	2246
	safe casts	0/901	194/901	311/901	**323/901**
	time (sec)	66	140	400	6+73
pmd	reachables	10558	10219	10193	10554
	call-graph (K)	11/56	124/569	158/1031	35/148
	points-to (K)	3591	4112	7065	4505
	safe casts	0/1076	175/1076	249/1076	**292/1076**
	time (sec)	118	158	332	9+139
batik	reachables	11210	10993	10988	11205
	call-graph (K)	11/61	82/1737	110/2139	20/306
	points-to (K)	2501	7194	10126	3479
	safe casts	0/976	266/976	306/976	**353/976**
	time (sec)	100	383	391	14+152
fop	reachables	27331	26315	26280	27272
	call-graph (K)	28/190	194/3710	329/6646	61/800
	points-to (K)	19549	22854	47296	20491
	safe casts	0/1487	329/1487	455/1487	**567/1487**
	time (sec)	433	753	1308	20+558

6.0.5 Copy Constant Propagation

Each handler is designed to handle a pre-defined set of message types. Thus sending a message to a handler not designed to handle its message type is a design bug. Running a demand-driven copy constant propagation [31, 32] at the reading of the `what` field (line 3) to enumerate all possible message types read by the handler is an approach to detect such bugs statically. Such a client analysis is evaluated in Section 6.0.5.

The copy constant propagation analysis is part of a tool developed to statically analyze the Android framework. Because constants are propagated through the heap, the effectiveness of the bug detection depends on precise points-to information. Since building precise points-to information requires deep context-sensitivity, which poses a significant challenge to the scalability of the analysis on large code base like the Android framework as illustrated in Section 1.1. To solve this problem we initiate Clipper at the reading of the `what` field (e.g. with the query access path i at line 3 in Fig. 1.6) to identify the relevant program elements and to handle them context-sensitively for precision, and handle the rest of the framework context-insensitively for scalability.

In this experiment, the package "`android.os`" (where the relevant library classes "`android.os.Message`" and "`android.os.Handler`" are defined) is added to the list of library packages to prevent jumping, as explained in Section 4.0.4.

We tested DynaSens on Android framework version 2.3.7_r1 (introduced in Section 1.1) with a harness generated with Droidel [33]. The precision is measured as the number of *safe* handlers, i.e., all possible message types handled are defined for the handler. Variants of DynaSens configured with context-sensitivities of depths up to 4 are tested. The results are presented in Table 6.3. Unlike the uniformly context-sensitive analyses in previous work [2], whose cost grows exponentially in context depth, all cost metrics of DynaSens grow much more slowly because of its accurate selective context sensitivity. Meanwhile, the precision continues to improve as the context depth becomes deeper, and all message handling sites are proven safe at depth 4, which would be intractable using a uniformly context sensitive analysis.

Table 6.3.: Results for demand-driven copy constant propagation. The “Reachables”, “Call-Graph”, “Points-To” and “Time” columns have the same meaning as in Table 6.2. The “Safe Handlers” column shows the number of message handlers proven safe by the copy constant propagation analysis.

Analysis	**Reachables**	**Call-Graph (K)**	**Points-To (K)**	**Safe Handlers**	**Time (sec)**
Insen	19 886	20/106	6494	0/95	147
2type+1H	18 860	117/1795	7356	35/95	516
1type1obj+1H	18 841	165/2264	10 620	35/95	574
DynaSens (2obj+1H)	19 717	39/179	6529	61/95	58+214
DynaSens (3obj+2H)	19 716	39/191	6603	61/95	58+228
DynaSens (4obj+3H)	19 716	40/189	6594	95/95	58+231

7. DYNASHAPE – A DEMAND-DRIVEN SHAPE ANALYSIS

Besides points-to analysis, Clipper can also be applied as a slicing analysis to more sophiscated and costly heap analyses such as shape analysis [34] to make the later demand-driven and hence precise as well as scalable.

In this chapter, the shape analysis based on the *Localized-heap Store-Less* ($\mathcal{LSL}$) semantics [12] (Section 7.1) is taken as an example to demonstrate utility of Clipper as a slicing analysis. The demand-driven variation of shape analysis called DynaSens is introduced in Section 7.2. The soundness of applying Clipper as a slicing analysis to the $\mathcal{LSL}$-semantics based shape analysis is proved in Section 7.3.

7.1 The $\mathcal{LSL}$ Semantics

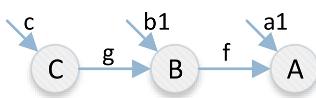
In traditional semantics, a memory heap is modeled as a graph where objects are represented as nodes and variables and fields are represented as edges. Such a model is called store-based heap model [12]. Different from traditional semantics, the $\mathcal{LSL}$ semantics is based on the storeless heap model [35] where an object is represented with an alias class, i.e. a set of access paths alias with each other, and a heap is represented with a set of objects disjoint with each other, i.e. a partition of all access paths within the heap, as defined in Fig. 7.1.

Example 6 *Fig. 7.2a shows a store-based model of the heap at certain execution state of the program in Fig. 7.3. Objects A, B, and C are modeled as nodes. Variables $a1$, $b1$, and c are modeled as arrows. Fields f and g are modeled as arrows between objects. In the corresponding storeless model of Fig. 7.2b, Objects are modeled as sets of heap access paths through which the objects are reached. For example, in the*

alias partition $\pi, \Pi \in APart \subset 2^{AClass}$
  s.t. $\forall A_1, A_2 \in \pi : A_1 \cap A_2 \neq \emptyset \Rightarrow A_1 = A_2$
alias class selector $[\cdot]_\pi : \bigcup \pi \to \pi$ for any $\pi \in APart$
  s.t. $\forall \alpha \in \bigcup \pi : \alpha \in [\alpha]_\pi \in \pi$
object or garbage (alias class) $\mathbb{o} \in \mathbb{Obj} \triangleq AClass$
garbage $\emptyset$
object $o \in Obj \triangleq \mathbb{Obj} \setminus \{\emptyset\}$
heap with garbage (alias partition) $\mathbb{H} \in \mathbb{Heap} \triangleq APart$
heap without garbage $H \in Heap \triangleq APart \cap 2^{Obj}$

Fig. 7.1.: The storeless heap model.

store-based model of Fig. 7.2a, Object A is reached via the access paths $a1$*,* $b1.f$*, and* $c.g.f$*. Hence A is represented by* $\{a1, b1.f, c.g.f\}$ *in Fig. 7.2b.*



(a) Store-based heap.

$$A = \{a1, b1.f, c.g.f\}$$
$$B = \{b1, c.g\}$$
$$C = \{c\}$$

(b) Storeless heap.

Fig. 7.2.: Example of store-based and storeless heap model.

The intraprocedural $\mathcal{LSL}$ semantics is introduced next. The example program in Fig. 7.3 and an execution trace in Fig. 7.4 is used to illustrate the intraprocedural transition rules. Without loss of generality, the execution of each assignment implicitly nullifies its left-hand side before running the assignment itself, i.e. "$x.f = y$;" is executed as "$x.f = null; x.f = y$;".

```
class A {
  int i;
}
```

```
class B {
  A f,g;
}
```

```
class C {
  B h;
}
```

```
void foo(A a1, B b1) {
  b1.f = a1;
  C c = new C();
  c.h = b1;
  B b2 = c.g;
  A a2 = new A();
  b2.f = a2;
}
```

Fig. 7.3.: Example code for illustrating intraprocedural $\mathcal{LSL}$.

In the original $\mathcal{LSL}$ semantics [12], an execution state σ is represented by a pair $\langle l, H\rangle$ consisting of the current statement label l and the current heap H. To help

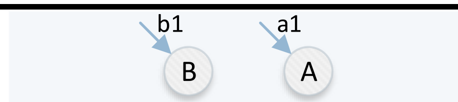
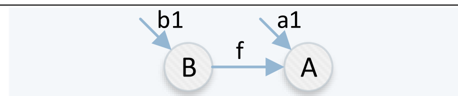
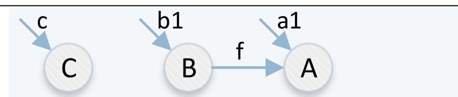
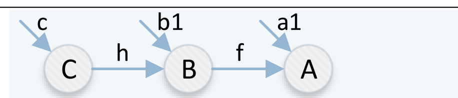
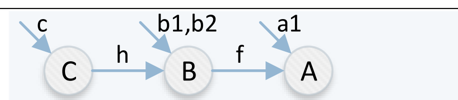
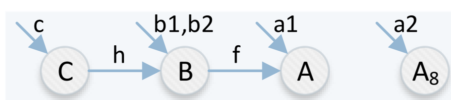
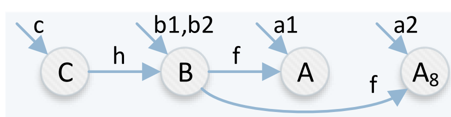
	Statement	Store-Based Heap	Storeless Heap
σ_1	2: b1.f = a1;		$A \rightarrow \{a1\}$ $B \rightarrow \{b1\}$
σ_2	3: c = new C();		$A \rightarrow \{a1, b1.f\}$ $B \rightarrow \{b1\}$
σ_3	4: c.h = b1;		$A \rightarrow \{a1, b1.f\}$ $B \rightarrow \{b1\}$ $C = \{c\}$
σ_4	5: b2 = c.h;		$A \rightarrow \{a1, b1.f, c.h.f\}$ $B \rightarrow \{b1, c.h\}$ $C = \{c\}$
σ_5	6: a2 = new A();		$A \rightarrow \{a1, b1.f, c.h.f\}$ $B \rightarrow \{b1, b2, c.h\}$ $C = \{c\}$
σ_6	7: b2.f = a2;		$A \rightarrow \{a1, b1.f, c.h.f\}$ $B \rightarrow \{b1, b2, c.h\}$ $A_8 = \{a2\}$ $C = \{c\}$
σ_7	8: exit;		$A \rightarrow \{a1\}$ $B \rightarrow \{b1, b2, c.h\}$ $A_8 = \{a2, b1.f, b2.f, c.h.f\}$ $C = \{c\}$

Fig. 7.4.: An execution trace of the program in Fig. 7.3. The upper part of each storeless heap encodes the *Tran* component and the lower part encodes the *Gen* component. For simplicity, the parameters of the *Tran* component are omitted, i.e. $A = \{a1\} \rightarrow \{a1, b1.f\}$ is encoded as $A \rightarrow \{a1, b1.f\}$

explaining the semantics, the current heap H is divided into two parts – the object transformer $Tran$ and the generated object set Gen, as defined in Fig. 7.5.

For objects that already exist at the entry of the current method, the object transformer $Tran$ maps their representation at the entry to their current representation. On the other hand, the generated object set Gen contains the objects created during the execution of the current method, i.e. those objects that do not exist at the entry of the current method.

object transformer:
$Tran \in Transformer = \mathbb{H} \rightarrow \mathbb{H}'$ for any $\mathbb{H}, \mathbb{H}' \in \mathbb{Heap}$
generated object set: $Gen \in Generated = \mathbb{Heap}$
state: $\sigma, \langle l, Tran, Gen \rangle \in \Sigma = Label \times Transformer \times Generated$
transition $\cdot \rightarrow \cdot : \Sigma \rightarrow \Sigma$

Fig. 7.5.: Execution state and transition of the $\mathcal{LSL}$ semantics.

Example 7 *In the execution trace of Fig. 7.4, Objects A and B are initially represented as $\{a1\}$ and $\{b1\}$, respectively, in the storeless heap. Within the heap at line 5, these objects are represented as $\{a1, b1, f, c.h.f\}$ and $\{b1, c.h\}$, respectively. Hence the Tran component at line 5 is $\big\{\{a1\} \rightarrow \{a1, b1, f, c.h.f\}, \{b1\} \rightarrow \{b1, c.h\}\big\}$. Since the object C does not exist at the entry of the current method, the Gen component at line 5 is $\big\{\{c\}\big\}$.*

The current heap H can be obtained by combining the $Tran$ component and the Gen component, i.e. $H = image(Tran) \cup Gen$.

The intraprocedural transition rule is given in Fig. 7.6 where each intraprocedural statement is modeled by the successor function $succ(\cdot)$ which, given the current state σ, returns a 3-tuple $\langle l', tran, gen \rangle$ consisting of

1. l': The label of the next statement to execute;

2. *tran*: An object transformer that maps the representation of each object in current state to its representation in the successor state;

3. *gen*: A set of objects generated by current statement.

Example 8 *In the execution trace of Fig. 7.4, the statement "4: c.h = b1;" at the state* σ_3 *is modeled by* $succ(\sigma_3) = \{\langle l', tran, gen\rangle\}$ *where*

$$\begin{cases} l' = 5 \\ tran = \big\{\{a1, b1.f\} \to \{a1, b1.f, c.h.f\}, \{b1\} \to \{b1, c.h\}, \{c\} \to \{c\}\big\} \\ gen = \{\} \end{cases}$$

Hence

$$\begin{cases} Tran' = & tran \circ Tran \\ = & tran \circ \big\{\{a1\} \to \{a1, b1.f\}, \{b1\} \to \{b1\}\big\} \\ = & \big\{\{a1\} \to \{a1, b1.f, c.h.f\}, \{b1\} \to \{b1, c.h\}\big\} \\ Gen' = & gen \cup map(tran)(Gen) \\ = & gen \cup map(tran)(\big\{\{c\}\big\}) \\ = & \big\{\{c\}\big\} \end{cases}$$

The transition rules for most intraprocedural statements are the same as the definition in the original $\mathcal{LSL}$ semantics. The object transformer *tran* for heap store statement "$y.f = x$" needs more explanation as this is where cycles can be created. For example, a cycle is created after executing "$y.f = x$" in the heap of Fig. 7.7b. Only two kinds of objects are affected by executing "$y.f = x$":

1. The object $[x]_H$, i.e. the object N_1;

2. Those objects that are reachable from $[x]_H$, e.g. the objects N_2 and A.

Only $[x]_H$ needs to be considered because given an object $o \in H$ reachable from $[x]_H$, i.e. $o = A \cup \bigcup\{[x]_H.\delta\}$, the new representation of o after executing "$y.f = x$"

intraprocedural state $\Sigma^{\mathtt{i}} \triangleq \{\langle l^{\mathtt{i}}, Tran^{\mathtt{i}}, Gen^{\mathtt{i}}\rangle \in \Sigma \mid stmt_l \text{ is intraprocedural}\}$
right-regularity closure $\rho_c(\cdot) : AClass \to AClass$
s.t. $\forall A \in AClass : \rho_c(A) = A \cup \bigcup\{\alpha.(\delta)^* \mid \alpha, \alpha.\delta \in A\}$

---

successor function $succ \in \Sigma^{\mathtt{i}} \to 2^{Label \times Transformer \times Generated}$
s.t. $\forall \langle l, Tran, Gen\rangle \in \Sigma^{\mathtt{i}} : \langle l', tran, gen\rangle \in succ(\langle l, Tran, Gen\rangle) \Leftrightarrow$
let $H = image(Tran) \cup Gen$ in

$$\wedge \begin{cases} l' = \begin{cases} l' & \text{if } l\text{: goto } l' \\ l_t & \text{if } l\text{: if } b\ l_t\ l_f \wedge True \sqsubseteq [\![b]\!](H) \\ l_f & l\text{: if } b\ l_t\ l_f \wedge False \sqsubseteq [\![b]\!](H) \\ l^{\mathtt{x}} & l\text{: return } z \\ l{+}1 & \text{otherwise} \end{cases} \\ tran = \begin{cases} \lambda o \in H.\ o \setminus x.\Delta & \text{if } l\text{: } x = null \\ \lambda o \in H.\ o \cup \{x.\delta \mid y.\delta \in o\} & \text{if } l\text{: } x = y \\ \lambda o \in H.\ o \cup \{ret.\delta \mid z.\delta \in o\} & \text{if } l\text{: return } z \\ \lambda o \in H.\ o \cup \{x.\delta \mid y.f.\delta \in o\} & \text{if } l\text{: } x = y.f \\ \lambda o \in H.\ o \setminus [y]_H.f.\Delta, \emptyset\rangle & \text{if } l\text{: } y.f = null \\ \lambda o \in H.\ o\ \cup\ \bigcup\{\rho_c([y]_H.f \cup [x]_H).\delta \mid x.\delta \in o\} & \\ & \text{if } l\text{: } y.f = x \\ \lambda o \in H.\ o & \text{otherwise} \end{cases} \\ gen = \begin{cases} \Big\{\{x\}\Big\} & \text{if } l\text{: } x = \text{new } t \\ \emptyset & \text{otherwise} \end{cases} \end{cases}$$

$$\dfrac{\begin{cases} stmt_l \text{ is intraprocedural} \\ \Diamond \begin{cases} \langle l', tran, gen\rangle \in succ(\langle l, Tran, Gen\rangle) \\ Tran' = tran \circ Tran \\ Gen' = gen \cup map(tran)(Gen) \end{cases} \end{cases}}{\langle l, Tran, Gen\rangle \to \langle l', Tran', Gen'\rangle}\ \text{INTRA}$$

Fig. 7.6.: Intraprocedural $\mathcal{LSL}$ semantics.

is $tran(o) = A \cup \bigcup\{[x]_{H'}.\delta\}$ where H' is the new representation of the heap after executing "$y.f = x$".

To model the effect of "$y.f = x$" on $[x]_H$, the set of access path $[y]_H.f = \{y.f, x.f.f\}$ is joined with $[x]_H = \{x\}$, yielding $\{y.f, x.f.f, x\} \subseteq [x]_{H'}$. According to right-regularity of storeless heap [36], both $x, x.f.f \in [x]_{H'}$ implies that

$$\{y.f, x.f.f, x\}(.f.f)^* \subseteq [x]_{H'}$$

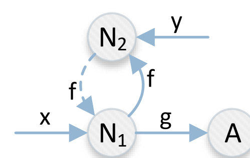
as imposed by the right-regularity closure ρ_c in Fig. 7.6. In this case,

$$\rho_c(\{y.f, x.f.f, x\}) = \{x(.f.f)^*, y.f(.f.f)^*\}$$

.

```
class N {
  N f;
  A g;
}
```

(a) Definition of class N.



(b) Store-based heap. The dashed arrow is the field to be added by "$y.f = x$".

$$N_1 = \{x\}$$
$$N_2 = \{y, x.f\}$$
$$A = \{x.g\}$$

(c) Storeless heap before executing "$y.f = x$".

$$N_1 = \{x(.f.f)^*, y.f(.f.f)^*\}$$
$$N_2 = \{y, x(.f.f)^*.f, y.f(.f.f)^*.f\}$$
$$A = \{x(.f.f)^*.g, y.f(.f.f)^*.g\}$$

(d) Storeless heap after executing "$y.f = x$".

Fig. 7.7.: Example of cycle created by heap store "$y.f = x$".

The interprocedural $\mathcal{LSL}$ semantics is introduced next. The example program in Fig. 7.8 and execution traces in Fig. 7.9, 7.10, 7.11, 7.12, 7.13, and 7.14 are used to illustrate the interprocedural transition rules.

```
1  void main() {
2    A am0 = new A();
3    B bm1 = foo1(am0);
4    A am1 = bm1.f;
5    int i1 = am1.i;
6    B bm2 = new B();
7    A am2 = new A();
8    bm2.f = am2;
9    foo2(bm2);
10   int i2 = am2.i;
11   A am3 = bm2.g;
12   int i3 = am3.i;
13 }
```

```
14 B foo1(A aa1) {
15   B bb1 = new B();
16   bb1.f = aa1;
17   bb1.g = aa1;
18   bar(bb1, 0, 1);
19   return bb1;
20 }
21 void foo2(B bb2) {
22   A aa3 = new A();
23   bb2.g = aa3;
24   bar(bb2, 2, 3);
25 }
```

```
26 void bar(B b, int j1, int j2) {
27   A a1 = b.f;
28   a1.i = j1;
29   A a2 = b.g;
30   a2.i = j2;
31 }
```

Fig. 7.8.: Example code for illustrating interprocedural $\mathcal{LSL}$.

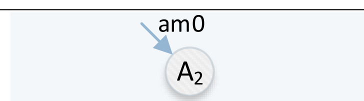
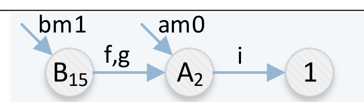
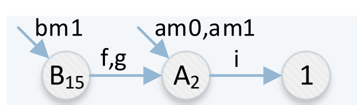
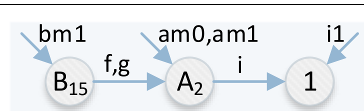
	Statement	Store-Based Heap	Storeless Heap
σ_1	2: am0 =new A();		
σ_2	3: bm1 =foo1(am0);		$A_2 = \{am0\}$
σ_3	4: am1 =bm1.f;		$1 = \{am0.i, bm1.f.i, bm1.g.i\}$ $A_2 = \{am0, bm1.f, bm1.g\}$ $B_{15} = \{bm1\}$
σ_4	5: i1 =am1.i;		$1 = \left\{\begin{array}{l} am0.i, am1.i, \\ bm1.f.i, bm1.g.i \end{array}\right\}$ $A_2 = \{am0, am1, bm1.f, bm1.g\}$ $B_{15} = \{bm1\}$
σ_5	6: bm2 =new B();		$1 = \left\{\begin{array}{l} am0.i, am1.i, \\ bm1.f.i, bm1.g.i \end{array}\right\}$ $A_2 = \{am0, am1, bm1.f, bm1.g\}$ $B_{15} = \{bm1\}$

Fig. 7.9.: An example trace of `main()`.

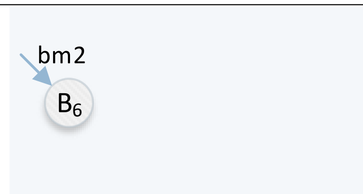
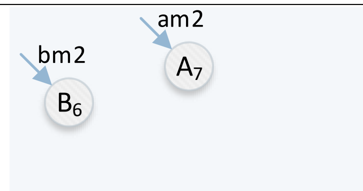
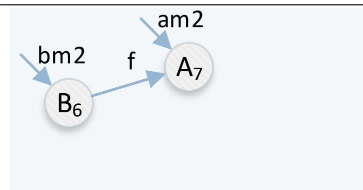
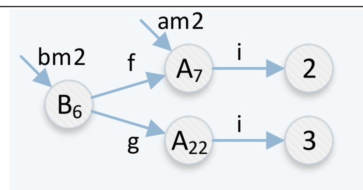
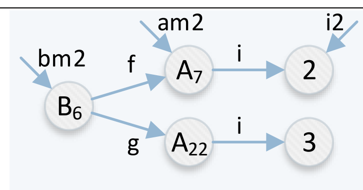
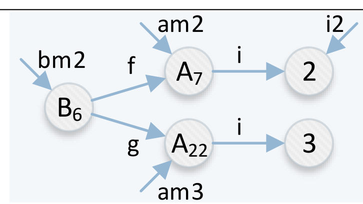
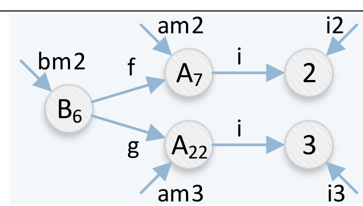
	Statement	Store-Based Heap	Storeless Heap
σ_5	6: bm2 =new B();		
σ_6	7: am2 = new A();		$B_6 = \{bm2\}$
σ_7	8: bm2.f = am2;		$A_7 = \{am2\}$ $B_6 = \{bm2\}$
σ_8	9: foo2(bm2);		$A_7 = \{am2, bm2.f\}$ $B_6 = \{bm2\}$
σ_9	10: i2 = am2.i;		$2 = \{am2.i, bm2.f.i\}$ $3 = \{bm2.g.i\}$ $A_7 = \{am2, bm2.f\}$ $A_{22} = \{bm2.g\}$ $B_6 = \{bm2\}$
σ_{10}	11: am3 = bm2.g;		$2 = \{am2.i, bm2.f.i, i2\}$ $3 = \{bm2.g.i\}$ $A_7 = \{am2, bm2.f\}$ $A_{22} = \{bm2.g\}$ $B_6 = \{bm2\}$
σ_{11}	12: i3 = am3.i;		$2 = \{am2.i, bm2.f.i, i2\}$ $3 = \{am3.i, bm2.g.i\}$ $A_7 = \{am2, bm2.f\}$ $A_{22} = \{am3, bm2.g\}$ $B_6 = \{bm2\}$
σ_{12}	13: exit;		$2 = \{am2.i, bm2.f.i, i2\}$ $3 = \{am3.i, bm2.g.i, i3\}$ $A_7 = \{am2, bm2.f\}$ $A_{22} = \{am3, bm2.g\}$ $B_6 = \{bm2\}$

Fig. 7.10.: An example trace of `main()` (continued). Part of the heap generated in Fig. 7.9 is omitted.

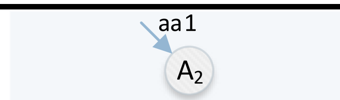
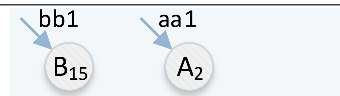
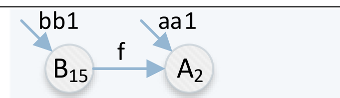
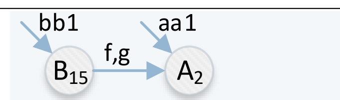
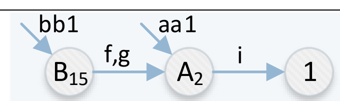
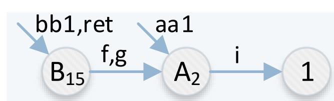
	Statement	Store-Based Heap	Storeless Heap
σ_{13}	15: bb1 = new B();		$A_2 \rightarrow \{aa1\}$
σ_{14}	16: bb1.f = aa1;		$A_2 \rightarrow \{aa1\}$ $B_{15} = \{bb1\}$
σ_{15}	17: bb1.g = aa1;		$A_2 \rightarrow \{aa1, bb1.f\}$ $B_{15} = \{bb1\}$
σ_{16}	18: bar(bb1, 0, 1);		$A_2 \rightarrow \{aa1, bb1.f, bb1.g\}$ $B_{15} = \{bb1\}$
σ_{17}	19: return bb1;		$A_2 \rightarrow \{aa1, bb1.f, bb1.g\}$ $1 = \{aa1.i, bb1.f.i, bb1.g.i\}$ $B_{15} = \{bb1\}$
σ_{18}	20: exit;		$A_2 \rightarrow \left\{ \begin{array}{l} aa1, bb1.f, bb1.g, \\ ret.f, ret.g \end{array} \right\}$ $1 = \left\{ \begin{array}{l} aa1.i, bb1.f.i, bb1.g.i, \\ ret.f.i, ret.g.i \end{array} \right\}$ $B_{15} = \{bb1, ret\}$

Fig. 7.11.: An example trace of `foo1(A aa1)`.

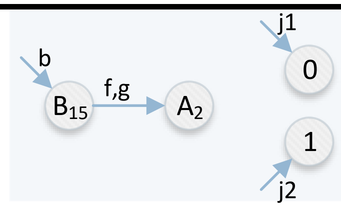
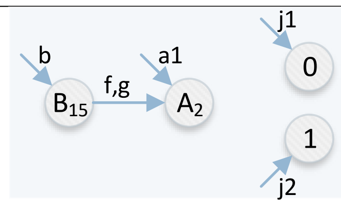
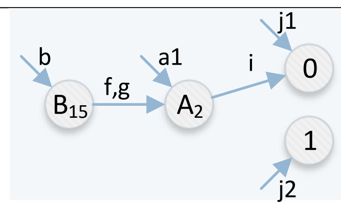
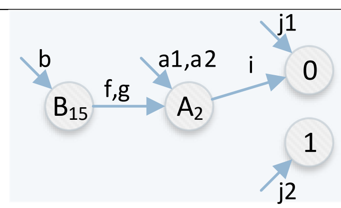
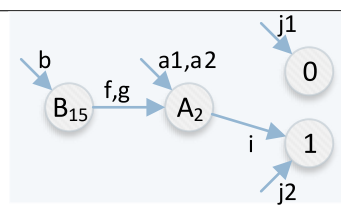
	Statement	Store-Based Heap	Storeless Heap
σ_{19}	27: a1 = b.f;		$0 \to \{j1\}$ $1 \to \{j2\}$ $A_2 \to \{b.f, b.g\}$ $B_{15} \to \{b\}$
σ_{20}	28: a1.i = j1;		$0 \to \{j1\}$ $1 \to \{j2\}$ $A_2 \to \{a1, b.f, b.g\}$ $B_{15} \to \{b\}$
σ_{21}	29: a2 = b.g;		$0 \to \{a1.i, b.f.i, b.g.i, j1\}$ $1 \to \{j2\}$ $A_2 \to \{a1, b.f, b.g\}$ $B_{15} \to \{b\}$
σ_{22}	30: a2.i = j2;		$0 \to \{a1.i, a2.i, b.f.i, b.g.i, j1\}$ $1 \to \{j2\}$ $A_2 \to \{a1, a2, b.f, b.g\}$ $B_{15} \to \{b\}$
σ_{23}	31: exit;		$0 \to \{j1\}$ $1 \to \{a1.i, a2.i, b.f.i, b.g.i, j2\}$ $A_2 \to \{a1, a2, b.f, b.g\}$ $B_{15} \to \{b\}$

Fig. 7.12.: An example trace of `bar(B b, int j1, int j2)`.

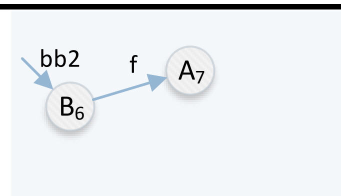
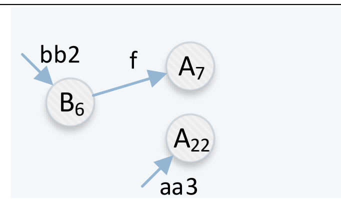
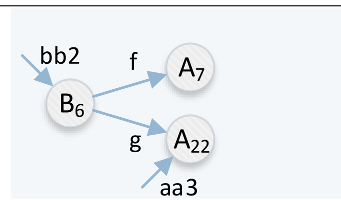
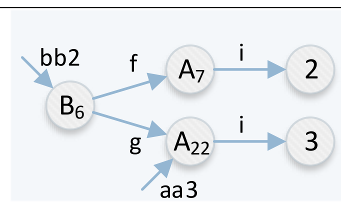
	Statement	Store-Based Heap	Storeless Heap
σ_{24}	22: aa3 = new A();		$A_7 \rightarrow \{bb2.f\}$ $B_6 \rightarrow \{bb2\}$
σ_{25}	23: bb2.g = aa3;		$A_7 \rightarrow \{bb2.f\}$ $B_6 \rightarrow \{bb2\}$ $A_{22} = \{aa3\}$
σ_{26}	24: bar(bb2, 2, 3);		$A_7 \rightarrow \{bb2.f\}$ $B_6 \rightarrow \{bb2\}$ $A_{22} = \{aa3, bb2.g\}$
σ_{27}	25: exit;		$A_7 \rightarrow \{bb2.f\}$ $B_6 \rightarrow \{bb2\}$ $2 = \{bb2.f.i\}$ $3 = \{aa3.i, bb2.g.i\}$ $A_{22} = \{aa3, bb2.g\}$

Fig. 7.13.: An example trace of `foo2(B bb2)`.

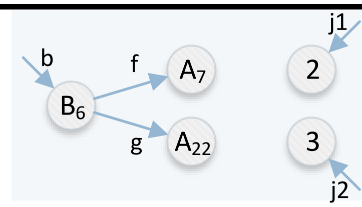
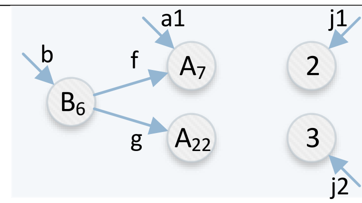
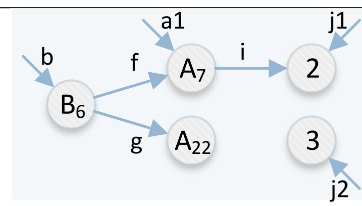
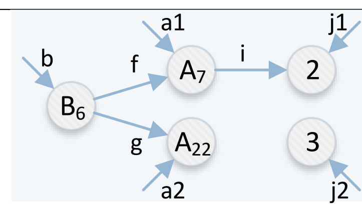
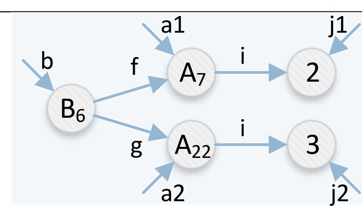
	Statement	Store-Based Heap	Storeless Heap
σ_{28}	27: a1 = b.f;		$2 \rightarrow \{j1\}$ $3 \rightarrow \{j2\}$ $A_7 \rightarrow \{b.f\}$ $A_{22} \rightarrow \{b.g\}$ $B_6 \rightarrow \{b\}$
σ_{29}	28: a1.i = j1;		$2 \rightarrow \{j1\}$ $3 \rightarrow \{j2\}$ $A_7 \rightarrow \{a1, b.f\}$ $A_{22} \rightarrow \{b.g\}$ $B_6 \rightarrow \{b\}$
σ_{30}	29: a2 = b.g;		$2 \rightarrow \{a1.i, b.f.i, j1\}$ $3 \rightarrow \{j2\}$ $A_7 \rightarrow \{a1, b.f\}$ $A_{22} \rightarrow \{b.g\}$ $B_6 \rightarrow \{b\}$
σ_{31}	30: a2.i = j2;		$2 \rightarrow \{a1.i, b.f.i, j1\}$ $3 \rightarrow \{j2\}$ $A_7 \rightarrow \{a1, b.f\}$ $A_{22} \rightarrow \{a2, b.g\}$ $B_6 \rightarrow \{b\}$
σ_{32}	31: exit;		$2 \rightarrow \{a1.i, b.f.i, j1\}$ $3 \rightarrow \{a2.i, b.f.i, j2\}$ $A_7 \rightarrow \{a1, b.f\}$ $A_{22} \rightarrow \{a2, b.g\}$ $B_6 \rightarrow \{b\}$

Fig. 7.14.: Another example trace of `bar(B b, int j1, int j2)`.

The $\mathcal{LSL}$ semantics is a natural (or big-step) one, i.e. each transition from a state at certain call site to another state at the corresponding return site is essentially an instantiation of certain exit trace of the callee, as specified in Fig. 7.15. An execution trace is a sequence of transitions where for each transition except the last one, the target of the transition is the source of the next transition, as define below. An exit trace is a trace where the target of the last transition is certain state at the exit statement "l: exit".

$$\text{trace } \sigma_0 \to \sigma_1 \to \cdots \to \sigma_k \in \mathit{Trace}$$
$$\text{exit trace } \mathit{Trace}^{\mathsf{x}} \triangleq \{\sigma^{\mathsf{e}} \to \cdots \to \langle l^{\mathsf{x}}, \mathit{Tran}^{\mathsf{x}}, \mathit{Gen}^{\mathsf{x}}\rangle \in \mathit{Trace} \mid l^{\mathsf{x}}\text{: exit}\}$$

Example 9 *The execution traces in Fig. 7.12 and 7.14 are two exit traces of the* `bar()` *method. The execution trace in Fig. 7.11 is an exit trace of the* `foo1()` *method and the execution trace in Fig. 7.13 is an exit trace of the* `foo2()` *method. The concatenation of traces in Fig. 7.9 and 7.10 is an exit trace of the* `main()` *method.*

Given a state $\langle l^{\mathsf{c}}, \mathit{Tran}^{\mathsf{c}}, \mathit{Gen}^{\mathsf{c}}\rangle$ at certain call site l^{c}: x=$p(y_0, \ldots, y_k)$ where the declaration of p is "$t\ p(t_0\ h_0, \ldots, t_k\ h_k)\{\mathit{body}_p\}$", the part of the heap visible to the callee consists of those objects reachable from arguments $y_0, \ldots, y_k$, as specified by the *passed* part H^{passed} in Fig. 7.15. The argument objects $[y_i]_{H^{\mathsf{c}}}$ are mapped to $\{h_i\}$ which act as base objects of callee's heap. Each object $o^{\mathsf{c}} \in H^{\mathit{passed}}$ needs to be "rebased" to map to callee's heap space, as specified by the operator $\cdot\downarrow$ which consists of the $\mathit{bind}_{\mathit{args}}$ operator and the $\mathit{sub}(\cdot)$ operator defined below.

$$\mathit{sub}(\cdot) : (H \to \mathit{AClass}) \to \mathit{Obj} \to \mathit{Obj} \text{ for certain } H \in \mathit{Heap}$$
$$\text{s.t. } \forall \mathit{bind} \in H \to \mathit{AClass}, o \in \mathit{Obj} :$$
$$\mathit{sub}(\mathit{bind})(o) \triangleq \bigcup_{o' \in H} \bigcup \{\mathit{bind}(o').\delta \mid \delta \in \Delta \wedge o'.\delta \subseteq o\}$$

$$
\frac{
\left\{
\begin{array}{l}
l^{\mathsf{c}}\colon x{=}p(y_0,\ldots,y_k) \\
\text{the declaration of } p \text{ is ``} t\ p(t_0\ h_0,\ldots,t_k\ h_k)\{body_p\}\text{''} \\
\spadesuit \left\{
\begin{array}{l}
l^{\mathsf{e}} \text{ is the entry label of } p \\
H^{\mathsf{c}} = image(Tran^{\mathsf{c}}) \cup Gen^{\mathsf{c}} \\
H^{args} = \{[y_i]_{H^{\mathsf{c}}} \mid 0 \le i \le k, [y_i]_{H^{\mathsf{c}}} \neq \emptyset\} \\
bind_{args} = \lambda o^{arg} \in H^{args}.\ \{h_i.\epsilon \mid 0 \le i \le k, y_i \in o^{arg}\} \\
\cdot\!\downarrow = \lambda o^{\mathsf{c}} \in H^{passed}.sub(bind_{args})(o^{\mathsf{c}}) \\
H^{passed} = \{o^{\mathsf{c}} \in H^{\mathsf{c}} \mid \exists o^{arg} \in H^{args}, \delta \in \Delta : o^{arg}.\delta \subseteq o^{\mathsf{c}}\} \\
H^{\mathsf{e}} = map(\cdot\!\downarrow)(H^{passed}) \\
Tran^{\mathsf{e}} = \lambda o^{\mathsf{e}} \in H^{\mathsf{e}}.\ o^{\mathsf{e}} \\
Gen^{\mathsf{e}} = \emptyset
\end{array}
\right. \\
\langle l^{\mathsf{e}}, Tran^{\mathsf{e}}, Gen^{\mathsf{e}}\rangle \to \cdots \to \langle l^{\mathsf{x}}, Tran^{\mathsf{x}}, Gen^{\mathsf{x}}\rangle \in Trace^{\mathsf{x}} \\
\clubsuit \left\{
\begin{array}{l}
l^{\mathsf{r}} = l^{\mathsf{c}} + 1 \\
H^{\mathsf{x}} = image(Tran^{\mathsf{x}}) \cup Gen^{\mathsf{x}} \\
H^{params} = map(\cdot\!\downarrow)(H^{args}) \\
H^{cp} = \{o^{cp} \in H^{passed} \mid \exists z \notin \overline{y}, \delta \in \Delta : z.\delta \in o^{cp}\} \\
H^{cpl} = map(\cdot\!\downarrow)(H^{cp}) \quad \text{// cut-point labels} \\
H^{\mathsf{x}root} = \cup \left\{
\begin{array}{l}
\{o^{\mathsf{x}ret} = [ret_p.\epsilon]_{H^{\mathsf{x}}}\} \\
H^{\mathsf{x}params} = map(Tran^{\mathsf{x}})(H^{params}) \\
H^{\mathsf{x}cp} = map(Tran^{\mathsf{x}})(H^{cpl})
\end{array}
\right. \\
Bypass = \lambda o \in Obj.\{x.\delta \in o \mid \forall \delta_1.\delta_2 = \delta : x.\delta_1 \notin \bigcup H^{passed}\} \\
bind_{cp} = \lambda o^{cp} \in H^{cp}.(o^{cp}\!\downarrow) \\
bind_{ret} = \lambda o^{\mathsf{x}} \in H^{\mathsf{x}root}. \\
\qquad \cup \left\{
\begin{array}{ll}
\{x.\epsilon \mid o^{\mathsf{x}} = o^{\mathsf{x}ret}\} & (a) \\
\bigcup \left\{
\begin{array}{l}
Bypass \circ bind_{args}^{-1}(o^{\mathsf{e}}) \mid \\
\quad o^{\mathsf{e}} \in H^{params} \wedge Tran^{\mathsf{x}}(o^{\mathsf{e}}) = o^{\mathsf{x}}
\end{array}
\right\} & (b) \\
\bigcup \left\{
\begin{array}{l}
Bypass \circ bind_{cp}^{-1}(o^{\mathsf{e}}) \mid \\
\quad o^{\mathsf{e}} \in H^{cpl} \wedge Tran^{\mathsf{x}}(o^{\mathsf{e}}) = o^{\mathsf{x}}
\end{array}
\right\} & (c)
\end{array}
\right. \\
\cdot\!\uparrow = \lambda o^{\mathsf{x}} \in H^{\mathsf{x}}. \\
\qquad \bigcup_{o^{\mathsf{x}root} \in H^{\mathsf{x}root}} \left\{ \bigcup \rho_c(sub(bind_{ret})(o^{\mathsf{x}root})).\delta \;\middle|\; o^{\mathsf{x}root}.\delta \subseteq o^{\mathsf{x}} \right\} \\
tran = \lambda o^{\mathsf{c}} \in H^{\mathsf{c}}. \left\{
\begin{array}{ll}
o^{\mathsf{c}} & \text{if } o^{\mathsf{c}} \in H^{\mathsf{c}} \setminus H^{passed} \\
Tran^{\mathsf{x}}(o^{\mathsf{c}}\!\downarrow)\!\uparrow & \text{if } o^{\mathsf{c}} \in H^{passed}
\end{array}
\right. \\
gen = map(\cdot\!\uparrow)(Gen^{\mathsf{x}}) \\
Tran^{\mathsf{r}} = tran \circ Tran^{\mathsf{c}} \\
Gen^{\mathsf{r}} = gen \cup map(tran)(Gen^{\mathsf{c}})
\end{array}
\right.
\end{array}
\right.
}{
\langle l^{\mathsf{c}}, Tran^{\mathsf{c}}, Gen^{\mathsf{c}}\rangle \to \langle l^{\mathsf{r}}, Tran^{\mathsf{r}}, Gen^{\mathsf{r}}\rangle
}\ \text{INTER}
$$

Fig. 7.15.: Interprocedural $\mathcal{LSL}$ semantics.

Intuitively, for any object $o^{\mathsf{c}} \in H^{passed}$, if $[y_i]_{H^{\mathsf{c}}}.\delta \subseteq o^{\mathsf{c}}$, then $\{h_i\}.\delta \in o^{\mathsf{c}}\!\downarrow$, i.e. the δ part is "rebased" from the old "base" $[y_i]_{H^{\mathsf{c}}}$ (i.e. the argument object) to the new "base" $\{h_i\}$ (i.e. the parameter object).

Example 10 *In the trace of Fig. 7.10 for example, given the state σ_8 at the call site "9: foo2(bm2)", H^{passed} – the part of the heap visible and hence passed to the callee – consists of objects B_6 and A_7, i.e. those objects reachable from the call argument "bm2". On the other hand, objects B_{15}, A_2, and 1 (omitted in Fig. 7.10 but shown in Fig. 7.9) are not reachable from the call argument "bm2" and hence not part of H^{passed}. The argument object $B_6 = \{bm2\}$ is mapped to $\{bb2\}$ where "bb2" is the parameter of method* `foo2(B bb2)`. *The object $A_7 = \{am2, bm2.f\}$ is "rebased" to $\{bb2.f\}$. Hence the heap at the entry of the callee* `foo2(B bb2)` *is $H^{\mathsf{e}} = \big\{\{bb2.f\}, \{bb2\}\big\}$.*

Given an exit trace of the callee, the instantiation of the exit trace at the call site requires more explanation. Only three kinds of root objects ($H^{\mathsf{x}root}$) within the heap at the exit state (the state $\langle l^{\mathsf{x}}, Tran^{\mathsf{x}}, Gen^{\mathsf{x}}\rangle$ where l^{x}: *exit*) need to be mapped back to the heap space of the caller:

1. Those objects reachable from callee parameters;

2. Those objects reachable from returned variable *ret*;

3. Those objects reachable from cut-point-labels [12] defined below.

Definition 7.1.1 (Cut-Point and Cut-Point-Label) *Given a state $\langle l^{\mathsf{c}}, Tran^{\mathsf{c}}, Gen^{\mathsf{c}}\rangle$ at certain call site l^{c}: x=$p(y_0, \ldots, y_k)$, a cut-point is an object $o^{cp} \in H^{passed}$ which is also reachable from the non-argument variables. The set of cut-points is denoted H^{cp}. A cut-point-label $o^{cpl} \in H^{\mathsf{e}}$ satisfies $o^{cpl} = o^{cp}\!\downarrow$ for certain cut-point $o^{cp} \in H^{cp}$. The set of cut-point-labels is denoted H^{cpl}.*

Example 11 *In the trace of Fig. 7.10 for example, given the state σ_8 at the call site "9: foo2(bm2)", The object $A_7 = \{am2, bm2.f\}$ is reachable from non-argument* *variable "am2" and hence a cut-point. Thus the object* $\{am2, bm2.f\}\downarrow = \{bb2.f\}$ *in the callee's heap space is a cut-point-label.*

Similar to the design of $\cdot\downarrow$, where each object $o^{\mathsf{c}} \in H^{passed}$ is "rebased" to map to callee's heap space, in the design of $\cdot\uparrow$, each object $o^{\mathsf{x}} \in H^{\mathsf{x}}$ is "rebased" to map to caller's heap space. There are three kinds of new/old "base" pairs

1. The old "base" is $[ret]_{H^{\mathsf{x}}}$ and the new "base" is $\{x\}$ (case (a) of $bind_{ret}$);
2. The old "base" is $[h_i]_{H^{\mathsf{x}}}$ and the new "base" is $[y_i]_{H^{\mathsf{c}}}$ (case (b) of $bind_{ret}$);
3. The old "base" is $Tran^{\mathsf{x}}(o^{cpl})$ and the new "base" is o^{cp} where o^{cp} is a cut-point and o^{cpl} is the corresponding cut-point-label, i.e. $o^{cpl} = o^{cp}\downarrow$ (case (c) of $bind_{ret}$).

The effect of a method call can be modeled by the following two functions:

1. *tran*: An object transformer that maps the representation of each object in the heap H^{c} of the state before the method call to its representation in the heap H^{r} of the state after the method call;
2. *gen*: A set of objects generated by the callee.

Example 12 *Given the state σ_2 at the call site "3: bm1=foo1(am0)" in the trace of Fig. 7.9 and an exit trace $\sigma_{12} \to \cdots \to \sigma_{17}$ of the callee* `foo1(A aa1)` *in Fig. 7.11, the method call can be modeled by*

$$\begin{cases} tran = \{\{am0\} \to \{am0, bm1.f, bm1.g\}\} \\ gen = \{\{am0.i, bm1.f.i, bm1.g.i\}, \{bm1\}\} \end{cases}$$

Hence

$$\begin{cases} Tran^{\mathsf{r}} = & tran \circ Tran^{\mathsf{c}} = tran \circ \{\} = \{\} \\ Gen^{\mathsf{r}} = & gen \cup map(tran)(Gen^{\mathsf{c}}) \\ = & gen \cup map(tran)(\{\{am0\}\}) \\ = & \{\{am0.i, bm1.f.i, bm1.g.i\}, \{am0, bm1.f, bm1.g\}, \{bm1\}\} \end{cases}$$

Similarly, given the state σ_8 at the call site "9: foo2(bm2)" in the trace of Fig. 7.10 and an exit trace $\sigma_{23} \to \cdots \to \sigma_{26}$ of the callee `foo2(B bb2)` *in Fig. 7.13, the method call can be modeled by*

$$\begin{cases} tran = \big\{\{am2, bm2.f\} \to \{am2, bm2.f\}, \{bm2\} \to \{bm2\}\big\} \\ gen = \big\{\{am2.i, bm2.f.i\}, \{bm2.g.i\}, \{bm2.g\}\big\} \end{cases}$$

Hence

$$\begin{cases} Tran^r = & tran \circ Tran^c = tran \circ \{\} = \{\} \\ Gen^r = & gen \cup map(tran)(Gen^c) \\ = & gen \cup map(tran)(\big\{\{am2, bm2.f\}, \{bm2\}\big\}) \\ = & \big\{\{am2.i, bm2.f.i\}, \{bm2.g.i\}, \{bm2.g\}, \{am2, bm2.f\}, \{bm2\}\big\} \end{cases}$$

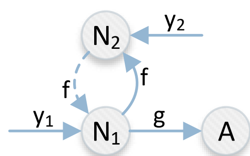
The senario where cycles are created by method calls needs furthur explanation. If cycles are created when merging the instantiation of callee's heap with caller's heap, only the root objects ($H^{\text{x}root}$) need to be considered because these objects are the new "bases" on which all callee's objects are "rebased", as specified in $\cdot\uparrow$.

For example, a cycle is created after executing "$x = p(y1, y2)$" in the heap of Fig. 7.16a. By imposing the right-regularity closure ρ_c on root objects (N_1 and N_2 in this example), all callee's objects reachable from the root objects are well-represented, as shown in Fig. 7.16d.

Example 13 *content...*

7.2 DynaShape

Since the $\mathcal{LSL}$ heap model is also access-path based, it is natural to apply Clipper as a slicing analysis to implement a demand-driven shape analysis based on $\mathcal{LSL}$ semantics. The demand-driven shape analysis specified here is called DynaShape.



(a) Store-based heap. The dashed arrow is the field to be added by "$x = p(y1, y2)$".

```
X p(N h1, N h2) {
  h2.f = h1;
  ...
}
```

(b) Definition of method `p`.

$$
\begin{aligned}
N_1 &= \{y1\} \\
N_2 &= \{y1.f, y2\} \\
A &= \{y1.g\}
\end{aligned}
$$

(c) Storeless heap before executing "$x = p(y1, y2)$".

$$
\begin{aligned}
N_1 &= \{y1(.f.f)^*, y2.f(.f.f)^*\} \\
N_2 &= \{y1(.f.f)^*.f, y2, y2.f(.f.f)^*.f\} \\
A &= \{y1(.f.f)^*.g, y2.f(.f.f)^*.g\}
\end{aligned}
$$

(d) Storeless heap after executing "$x = p(y1, y2)$".

Fig. 7.16.: Example of cycle created by method call "$x = p(y1, y2)$".

Assume a shape analysis is needed to track flow of values, which is a typical demand in many analyses such as *Typestate* analysis [37]. One major challenge in such analyses is tracking data flow through the heap precisely. Although shape analyses can model the heap precisely, it is unnecessary (and expensive) to analyze the whole program exhaustively.

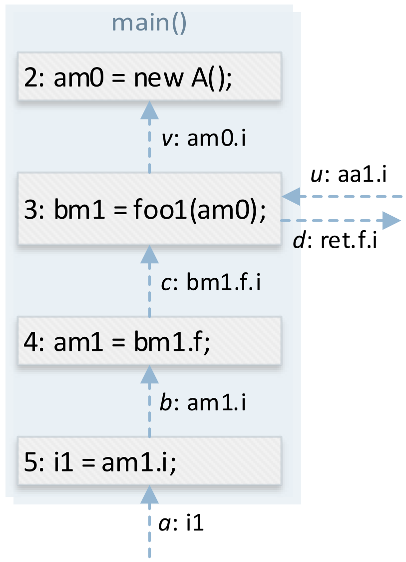
Example 14 *In the program of Fig. 7.8 it is unnecessary to analyze statements handling variable $j2$ in method* `bar()` *to track the values flow to variable $i2$ in method* `main()`*. On the other hand, aliasing poses challenges when deciding which part of the program is omittable. For example, in the program of Fig. 7.8 it is necessary to analyze statements handling variable $j2$ in method* `bar()` *to track the values flow to variable $i1$ in method* `main()` *due to aliasing.*

Such nuance caused by aliasing can be detected and handled precisely by Clipper*, as shown by the slicing result in Fig. 7.17 and 7.18. In the slicing analysis with respect to $i1$ in Fig. 7.17a 7.17b 7.17c 7.17d, statements related to both variables $j1$ and $j2$ in method* `bar()` *are included in the generated slice (Fig. 7.17c and 7.17d). On the other hand, in the slicing analysis with respect to $i2$ in Fig. 7.18a 7.18b 7.17c, only statements related to variable $j2$ in method* `bar()` *are included in the generated slice (Fig. 7.17d).*

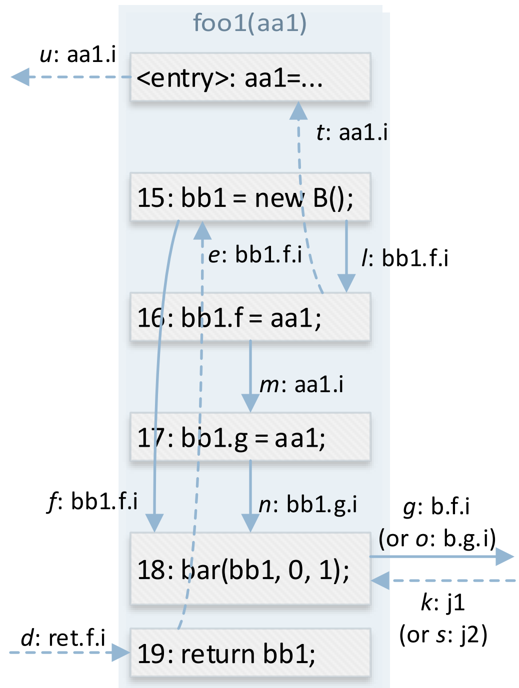
Given a slice $Slice \in 2^{Label}$ (i.e. part of the target program encoded as a set of statement labels) generated with respect to certain slicing criteria α, a parametric shape analysis parameterized with respect to *Slice* (as specified in Fig. 7.19) can be tailored to focus on statements in *Slice* only and ignore the rest.

If a statement is in *Slice*, the statement is handled normally according to *Lsl* semantics (rules INTRA$^{\mathcal{S}+}$ and INTER$^{\mathcal{S}+}$). Otherwise, the statement is handled as a no-op (rules INTRA$^{\mathcal{S}-}$ and INTER$^{\mathcal{S}-}$).

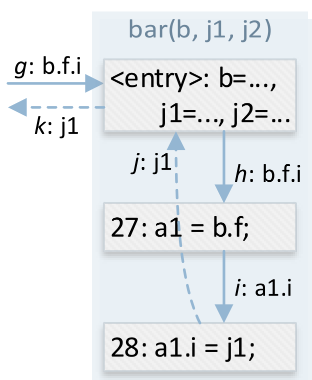
Example 15 *Given the slice with respect to $i1$ in method* `main()` *(Fig. 7.17a 7.17b) 7.17c 7.17d), the tailored execution traces are those in Fig. 7.9 7.11 7.12.*



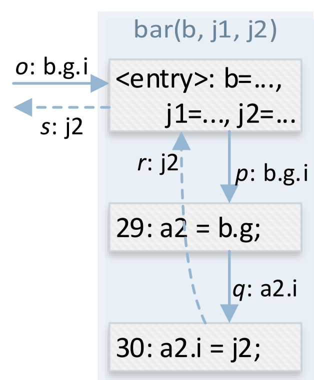
(a) The derivation for facts of the form $A(i2, \_)$ on `main()`.



(b) The derivation for facts of the form $A(ret.f.i, \_)$ on `foo1()`.

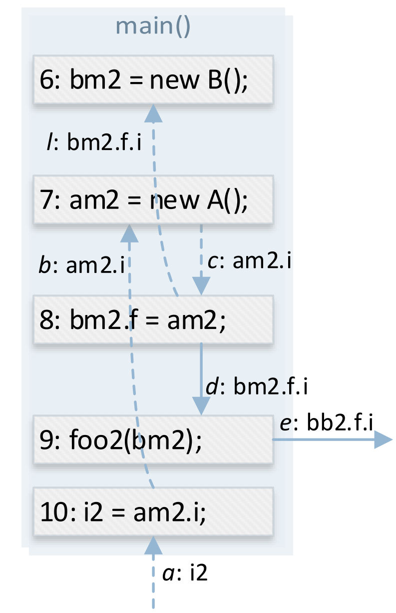


(c) The derivation for facts of the form $A(b.f.i, \_)$ on `bar()`.

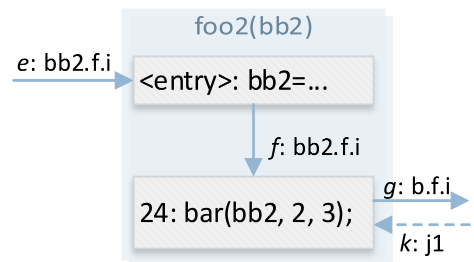


(d) The derivation for facts of the form $A(b.g.i, \_)$ on `bar()`.

Fig. 7.17.: Running Clipper with $i1$ as the slicing criteria.



(a) The derivation for facts of the form $A(i2, \_)$ on `main()`.



(b) The derivation for facts of the form $A(bb2.f.i, \_)$ on `foo2()`.

Fig. 7.18.: Running Clipper with $i2$ as the slicing criteria.

$$\frac{l \in \mathit{Slice} \qquad \begin{cases} \mathit{stmt}_l \text{ is intraprocedural} \\ \diamondsuit \end{cases}}{\sigma{=}\langle l, \mathit{Tran}, \mathit{Gen}\rangle \xrightarrow{\mathcal{S}} \langle l', \mathit{Tran}', \mathit{Gen}'\rangle}\ \text{INTRA}^{\mathcal{S}+}$$

$$\frac{l \notin \mathit{Slice} \qquad \begin{cases} \mathit{stmt}_l \text{ is intraprocedural} \\ l' \in \mathit{succ}(\sigma) \\ \mathit{Tran}' = \mathit{Tran} \\ \mathit{Gen}' = \mathit{Gen} \end{cases}}{\sigma{=}\langle l, \mathit{Tran}, \mathit{Gen}\rangle \xrightarrow{\mathcal{S}} \langle l', \mathit{Tran}', \mathit{Gen}'\rangle}\ \text{INTRA}^{\mathcal{S}-}$$

(a) Intraprocedural transitions.

$$\frac{l^{\mathtt{c}} \in \mathit{Slice} \qquad \begin{cases} l^{\mathtt{c}}\!: x{=}p(y_0, \ldots, y_k) \\ \text{the declaration of } p \text{ is ``} t\ p(t_0\ h_0, \ldots, t_k\ h_k)\{\mathit{body}_p\}\text{''} \\ \spadesuit \\ \langle l^{\mathtt{e}}, \mathit{Tran}^{\mathtt{e}}, \mathit{Gen}^{\mathtt{e}}\rangle \xrightarrow{\mathcal{S}} \cdots \xrightarrow{\mathcal{S}} \langle l^{\mathtt{x}}, \mathit{Tran}^{\mathtt{x}}, \mathit{Gen}^{\mathtt{x}}\rangle \in \mathit{Trace}^{\mathtt{x}} \\ \clubsuit \end{cases}}{\langle l^{\mathtt{c}}, \mathit{Tran}^{\mathtt{c}}, \mathit{Gen}^{\mathtt{c}}\rangle \xrightarrow{\mathcal{S}} \langle l^{\mathtt{r}}, \mathit{Tran}^{\mathtt{r}}, \mathit{Gen}^{\mathtt{r}}\rangle}\ \text{INTER}^{\mathcal{S}+}$$

$$\frac{l^{\mathtt{c}} \notin \mathit{Slice} \qquad \begin{cases} l^{\mathtt{c}}\!: x{=}p(y_0, \ldots, y_k) \\ l^{\mathtt{r}} = l^{\mathtt{c}} + 1 \\ \mathit{Tran}^{\mathtt{r}} = \mathit{Tran}^{\mathtt{c}} \\ \mathit{Gen}^{\mathtt{r}} = \mathit{Gen}^{\mathtt{c}} \end{cases}}{\langle l^{\mathtt{c}}, \mathit{Tran}^{\mathtt{c}}, \mathit{Gen}^{\mathtt{c}}\rangle \xrightarrow{\mathcal{S}} \langle l^{\mathtt{r}}, \mathit{Tran}^{\mathtt{r}}, \mathit{Gen}^{\mathtt{r}}\rangle}\ \text{INTER}^{\mathcal{S}-}$$

(b) Interprocedural transitions.

Fig. 7.19.: DynaShape analysis.

Given the slice with respect to i2 in method **`main()`** *(Fig. 7.18a 7.18b) 7.17c), the tailored execution traces are those in Fig. 7.20. Compared with the complete traces in Fig. 7.10 7.13 7.14, the tailored ones are much simplified.*

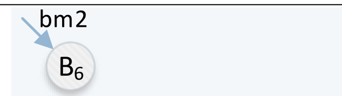
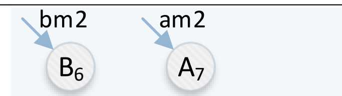
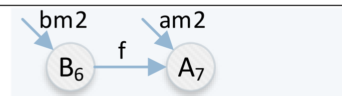
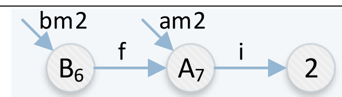
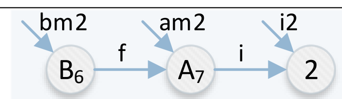
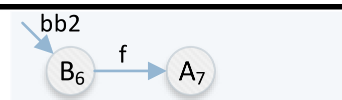
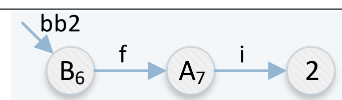
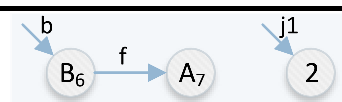
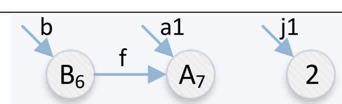
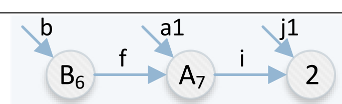
	Statement	Store-Based Heap	Storeless Heap
σ_5	6: bm2 =new B();		
σ_{33}	7: am2 = new A();		$B_6 = \{bm2\}$
σ_{34}	8: bm2.f = am2;		$A_7 = \{am2\}$ $B_6 = \{bm2\}$
σ_{35}	9: foo2(bm2);		$A_7 = \{am2, bm2.f\}$ $B_6 = \{bm2\}$
σ_{36}	10: i2 = am2.i;		$2 = \{am2.i, bm2.f.i\}$ $A_7 = \{am2, bm2.f\}$ $B_6 = \{bm2\}$
σ_{37}	13: exit;		$2 = \{am2.i, bm2.f.i, i2\}$ $A_7 = \{am2, bm2.f\}$ $B_6 = \{bm2\}$
σ_{38}	24: bar(bb2, 2, 3);		$A_7 \rightarrow \{bb2.f\}$ $B_6 \rightarrow \{bb2\}$
σ_{39}	25: exit;		$A_7 \rightarrow \{bb2.f\}$ $B_6 \rightarrow \{bb2\}$ $2 = \{bb2.f.i\}$
σ_{40}	27: a1 = b.f;		$2 \rightarrow \{j1\}$ $A_7 \rightarrow \{b.f\}$ $B_6 \rightarrow \{b\}$
σ_{41}	28: a1.i = j1;		$2 \rightarrow \{j1\}$ $A_7 \rightarrow \{a1, b.f\}$ $B_6 \rightarrow \{b\}$
σ_{42}	31: exit;		$2 \rightarrow \{a1.i, b.f.i, j1\}$ $A_7 \rightarrow \{a1, b.f\}$ $B_6 \rightarrow \{b\}$

Fig. 7.20.: A tailored trace focusing on data flow to $i2$.

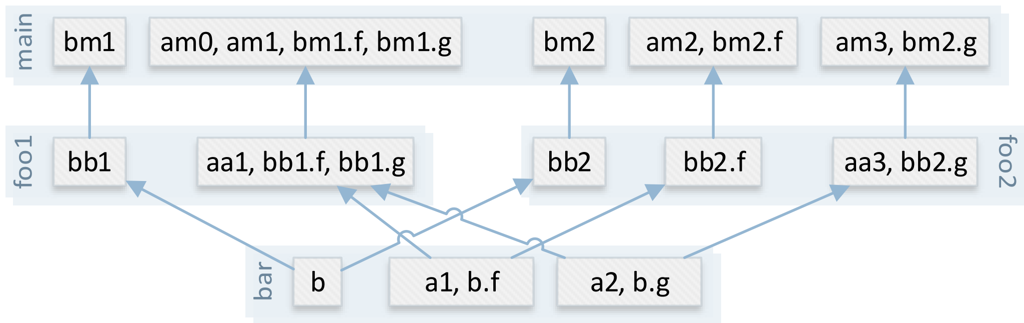


Fig. 7.21.: Part of the alias partition built from bottom up according to the program of Fig. 7.8. The arrows represent instantiation of callee's alias classes in caller's alias partition.

7.3 Soundness of Clipper as a Slicing Analysis

A proof of the soundness of Clipper as a slicing analysis with respect to the $\mathcal{LSL}$ semantics is given in this section. The proof is outlined as follows: Given a program, the deduction rules of Clipper essentially build alias classes (encoded as an alias relation in Section 7.3.1) from bottom up with respect to the call graph. An instrumented small-step $\mathcal{LSL}^{\mathcal{I}}$ semantics is given (in Section 7.3.2), which augment the execution state with an alias partition Π (i.e. a set of alias classes) maintained from top down with respect to the call graph. The augmented alias partition summarizes the alias classes inferred by Clipper from bottom up. The initial alias partition Π^0 is the top one (i.e. at the entry method, typically `main()`) built from bottom up by Clipper. This chapter tries to prove the invariance that at any time during the execution of a program, the current heap H is over-approximated by the current alias partition Π.

Example 16 *Fig. 7.21 shows some of the alias partitions built by* Clipper *from bottom up. Starting from the top-level alias partition of method* `main()`*, the alias partitions built from top down by the* $\mathcal{LSL}^{\mathcal{I}}$ *semantics is shown in Fig. 7.22. It is verifiable that any heap occured in the traces of Fig. 7.9 7.10 7.11 7.13 7.12 7.14 is over-approximated by the corresponding partition in Fig. 7.22.*

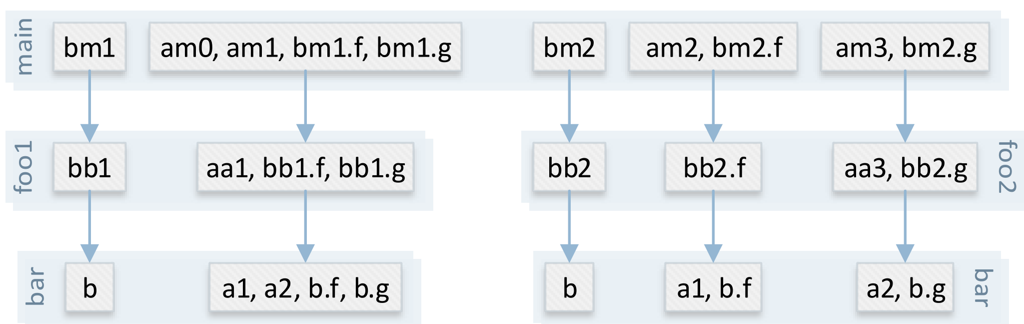


Fig. 7.22.: Part of the alias partition built from top down according to the program of Fig. 7.8. The arrows represent instantiation of caller's alias classes in callee's alias partition.

7.3.1 Clipper as a Deductive System for Alias Relations

The fact $A(\alpha, \beta)$ derived by CLIPPER, previously interpreted as "β belongs to the alias class of α" (i.e. $\beta \in [\alpha]$), can also be intepreted as "α and β are aliases for each other" (i.e. $\langle\alpha, \beta\rangle \in R$ for an alias relation R defined below)

$$\begin{aligned} &\text{alias relation (on } AP\text{) } R \in ARel = 2^{AP \times AP} \\ &\quad \text{s.t. } R \text{ is closed under reflexivity, symmetry, and transitivity.} \end{aligned}$$

Therefore given a program *prog*, CLIPPER can also be encoded as a set of alias facts and rules $[\![prog]\!]^{\mathbb{R}}$ for deriving a set of alias relations $\overline{R}$ (one for each method), as shown in Fig. 7.23.

Since an alias relation is an equivalence relation (i.e. is reflexive, symmetric, and transitive), an alias relation R can be equivalently encoded as an alias partition (via the operator $\Pi(\cdot)$ defined below) and vise versa [38].

$$\begin{aligned} &\Pi(\cdot) : ARel \to APart \\ &\quad \text{s.t. } \forall R \in ARel : \Pi(R) \triangleq \{\{\beta \mid \langle\alpha, \beta\rangle \in R\} \mid \alpha \in AP\} \end{aligned}$$

Hence

$$\forall R_p \in \overline{R} : \forall\langle\alpha, \beta\rangle \in R_p : \begin{cases} [\alpha]_{\Pi(R_p)} = [\beta]_{\Pi(R_p)} \\ \alpha \in [\beta]_{\Pi(R_p)} \\ \beta \in [\alpha]_{\Pi(R_p)} \end{cases}$$

Proposition 7.3.1 *For any program prog and the set of alias relations $\overline{R}$ derived from $[\![prog]\!]^{\mathbb{R}}$, any $R_p \in \overline{R}$ is closed under right-regularity, i.e.,*

$$\forall R_p \in \overline{R}, \delta \in \Delta : \forall\langle\alpha, \beta\rangle \in R_p : \langle\alpha.\delta, \beta.\delta\rangle \in R_p$$

$$\text{let } EQ = \cup \begin{cases} \bigcup_{R_p \in \overline{R}} \{R_p(x.\delta, x.\delta) \mid x \in Var_p \wedge \delta \in \Delta\} & \text{// reflexivity} \\ \{R(\alpha, \beta) \text{ :- } R_p(\beta, \alpha)\} & \text{// symmetry} \\ \{R(\alpha_1, \alpha_3) \text{ :- } R(\alpha_1, \alpha_2), R(\alpha_2, \alpha_3)\} & \text{// transitivity} \end{cases} \text{ in}$$

$$[\![prog]\!]^{\mathbb{R}} = EQ \cup [\![cdecl_1, ..., cdecl_k]\!]^{\mathbb{R}} = EQ \cup \bigcup_{1 \le i \le k} [\![cdecl_i]\!]^{\mathbb{R}}$$

$$[\![cdecl]\!]^{\mathbb{R}} = [\![\text{class } t \ \{\overline{fdecl} \ mdecl_1, ..., mdecl_k\}]\!]^{\mathbb{R}} = \bigcup_{1 \le i \le k} [\![mdecl_i]\!]^{\mathbb{R}}$$

$$[\![mdecl]\!]^{\mathbb{R}} = [\![t \ p(\overline{t \ h}) \ \{stmt_1, ..., stmt_k\}]\!]^{\mathbb{R}} = \bigcup_{1 \le i \le k} [\![stmt_i]\!]^{\mathbb{R}}$$

$$[\![stmt]\!]^{\mathbb{R}} =$$

$$\begin{cases} \{R_p(x.\delta, y.\delta) \mid \delta \in \Delta\} & \text{if stmt is } l{:}x = y \text{ in method } p \\ \{R_p(x.\delta, y.f.\delta) \mid \delta \in \Delta\} & \text{if stmt is } l{:}x = y.f \text{ in method } p \\ \{R_p(x.\delta, y.f.\delta) \mid \delta \in \Delta\} & \text{if stmt is } l{:}y.f = x \text{ in method } p \\ \cup \begin{cases} \{R_q(y_i.\delta_1, y_j.\delta_2) \text{ :- } R_p(h_i.\delta_1, h_j.\delta_2) \mid 0 \le i, j \le k\} \\ \{R_q(y_i.\delta_1, x.\delta_2) \text{ :- } R_p(h_i.\delta_1, z.\delta_2) \mid 0 \le i \le k\} \\ \{R_q(x.\delta_1, x.\delta_2) \text{ :- } R_p(ret_p.\delta_1, ret_p.\delta_2)\} \end{cases} & \\ & \text{if} \begin{cases} \text{stmt is } l{:}x{=}p(y_0, \ldots, y_k) \text{ in method } q \\ \text{declaration of } p \text{ is ``} t \ p(t_0 \ h_0, \ldots, t_k \ h_k)\{body_p\}\text{''} \end{cases} \\ \emptyset & \text{otherwise} \end{cases}$$

Fig. 7.23.: Basic facts and derivation rules for defining alias relation $\overline{R}$.

Proof For any $R_p \in \overline{R}$ and any pair of access paths $\langle \alpha, \beta \rangle \in R_p$, there is a derivation tree $Tree$ consisting of a set of basic facts and derivations from which the alias fact $R_p(\alpha, \beta)$ is derived. A derivation represents an instantiation of certain derivation rule.

Next the operator $\cdot.\cdot$ is defined as below:

$$\begin{cases} \forall R_p \in \overline{R} \text{ and } \alpha, \beta \in AP \text{ and } \delta \in \Delta : R_p(\alpha, \beta).\delta \triangleq R_p(\alpha.\delta, \beta.\delta) \\ \forall R_p, R_q \in \overline{R} \text{ and } \alpha_1, \alpha_2, \beta_1, \beta_2 \in AP \text{ and } \delta \in \Delta : \\ \quad (R_p(\beta_1, \beta_2) \text{ :- } R_q(\alpha_1, \alpha_2)).\delta \triangleq R_p(\beta_1.\delta, \beta_2.\delta) \text{ :- } R_q(\alpha_1.\delta, \alpha_2.\delta) \\ \forall R_p \in \overline{R} \text{ and } \alpha_1, \alpha_2, \alpha_3 \in AP \text{ and } \delta \in \Delta : \\ \quad (R_p(\alpha_1, \alpha_3) \text{ :- } R_p(\alpha_1, \alpha_2), R_p(\alpha_2, \alpha_3)).\delta \triangleq \\ \quad R_p(\alpha_1.\delta, \alpha_3.\delta) \text{ :- } R_p(\alpha_1.\delta, \alpha_2.\delta), R_p(\alpha_2.\delta, \alpha_3.\delta) \end{cases}$$

From the definition of $[\![\cdot]\!]^{\mathbb{R}}$, it follows that

$$\forall \delta \in \Delta : \begin{cases} \forall R_p \in \overline{R} \text{ and } \alpha, \beta \in AP : R_p(\alpha, \beta) \in [\![prog]\!]^{\mathbb{R}} \Rightarrow R_p(\alpha, \beta).\delta \in [\![prog]\!]^{\mathbb{R}} \\ \forall R_p, R_q \in \overline{R} \text{ and } \alpha_1, \alpha_2, \beta_1, \beta_2 \in AP \text{ and } rule \in [\![prog]\!]^{\mathbb{R}} : \\ \quad R_q(\beta_1, \beta_2) \text{ :- } R_p(\alpha_1, \alpha_2) \text{ instantiates } rule \Rightarrow \\ \quad (R_q(\beta_1, \beta_2) \text{ :- } R_p(\alpha_1, \alpha_2)).\delta \text{ instantiates } rule \\ \forall R_p \in \overline{R} \text{ and } \alpha_1, \alpha_2, \alpha_3 \in AP \text{ and } rule \in [\![prog]\!]^{\mathbb{R}} : \\ \quad R_p(\alpha_1, \alpha_3) \text{ :- } R_p(\alpha_1, \alpha_2), R_p(\alpha_2, \alpha_3) \text{ instantiates } rule \Rightarrow \\ \quad (R_p(\alpha_1, \alpha_3) \text{ :- } R_p(\alpha_1, \alpha_2), R_p(\alpha_2, \alpha_3)).\delta \text{ instantiates } rule \end{cases}$$

It follows that for any $\delta \in \Delta$, $\cdot.\delta$ is a homomorphism, i.e., for any alias fact $R_p(\alpha, \beta)$ derivable by $Tree$, $R_p(\alpha.\delta, \beta.\delta)$ is also derivable by $Tree' = \{elem.\delta \mid elem \in Tree\}$. Hence any $R_p \in \overline{R}$ is closed under right-regularity.

■

Corollary 7.3.2 *Given alias relations $\overline{R}$ derived from $[\![prog]\!]^{\mathbb{R}}$ of program prog,*

$$\forall R_p \in \overline{R}, A \in AClass : \forall A' \in \Pi(R_p) : A \subseteq A' \Rightarrow \rho_c(A) \subseteq A'$$

Proof It follows from the right-regularity of $\overline{R}$ that

$$\begin{aligned} \forall \alpha \in AP, \delta \in \Delta : \langle \alpha, \alpha.\delta \rangle \in R_p \\ &\Rightarrow \langle \alpha.\delta, \alpha.\delta.\delta \rangle \in R_p \\ &\Rightarrow \langle \alpha.\delta.\delta, \alpha.\delta.\delta.\delta \rangle \in R_p \\ &\Rightarrow \ldots \\ &\Rightarrow \alpha.(\delta)^* \in [\alpha]_{\Pi(R_p)} \end{aligned}$$

■

In the next section, the $\mathcal{LSL}$ semantics will be augmented with the alias relations $\overline{R}$ inferred by Clipper to prove the invariance mentioned at the beginning of this chapter.

7.3.2 $\mathcal{LSL}^{\mathcal{I}}$ – An Instrumented Small-Step $\mathcal{LSL}$ Semantics

In this section, a small-step (or stack-based) semantics $\mathcal{LSL}^{\mathcal{I}}$ augmented with the alias relations $\overline{R}$ inferred by Clipper will be used to prove the soundness of Clipper as a slicing analysis for the $\mathcal{LSL}$ semantics. There are two reasons to choose a small-step (or stack-based) semantics:

1. Interprocedural transitions (e.g. those from call sites of callers to entries of callees and those from exits of callees to return sites of callers) are well-defined in small-step semantics;

2. Stacks impose constraints on the structure of realizable traces, as defined in Fig. 7.26 and 7.27.

Therefore, two additional components are added to a $\mathcal{LSL}$ state σ to form a $\mathcal{LSL}^{\mathcal{I}}$ state $\hat{\sigma}$ – an alias relation R inferred by Clipper and a stack of pending call states S, as defined in Fig. 7.24.

$$\begin{aligned}&\mathcal{LSL}^{\mathcal{I}} \text{ stack: } S \in Stack = (Label \times Transformer \times Generated \times ARel)^*\\&\mathcal{LSL}^{\mathcal{I}} \text{ state: } \hat{\sigma}, \langle l, Tran, Gen, R, S\rangle \in \widehat{\Sigma} =\\&\quad Label \times Transformer \times Generated \times ARel \times Stack\end{aligned}$$

Fig. 7.24.: Execution state of the $\mathcal{LSL}^{\mathcal{I}}$ semantics.

The transition rules of $\mathcal{LSL}^{\mathcal{I}}$ semantics are specified in Fig. 7.25.

The intraprocedural transition rule (rule $\text{INTRA}^{\mathcal{I}}$) of the $\mathcal{LSL}^{\mathcal{I}}$ semantics are almost the same as that of the $\mathcal{LSL}$ semantics. The interprocedural transition rule of the $\mathcal{LSL}$ semantics is devided into two small-step style transition rules (rule $\text{CALL}^{\mathcal{I}}$ and $\text{RETURN}^{\mathcal{I}}$) of the $\mathcal{LSL}^{\mathcal{I}}$ semantics.

The call rule $\text{CALL}^{\mathcal{I}}$ pushes the current state onto the stack and compute the entry state $\langle l^{\mathsf{e}}, Tran^{\mathsf{e}}, Gen^{\mathsf{e}}\rangle$ and alias relation R^{d} for the callee. The entry state $\langle l^{\mathsf{e}}, Tran^{\mathsf{e}}, Gen^{\mathsf{e}}\rangle$ is computed in the same way as the $\mathcal{LSL}$ semantics while alias relation R^{d} for the callee is computed by first instantiating the alias relation R^{u} of the caller at the callee ($R^{\mathsf{d}}\downarrow_p^{l^{\mathsf{c}}}$), then extending it with the alias relation inferred by Clipper from bottom up ($R^{\mathsf{d}}\downarrow_p^{l^{\mathsf{c}}} \cup R_p$), and finally close it according to reflexivity, symmetry, transitiviy and right-regularity ($\rho_{rstc}(R^{\mathsf{u}}\downarrow_p^{l^{\mathsf{c}}} \cup R_p)$).

Example 17 *Given the caller alias relation (encoded as an alias partition)* $\Pi(R^{\mathsf{u}}) = \big\{\{bb1\}, \{aa1, bb1.f, bb1.g\}\big\}$ *in Fig. 7.22 for example, at the call site "18: bar(bb1,0,1)" in method* `foo1()` *of Fig. 7.8, the callee alias relation* R^{d} *is computed as follows:*

$$\Pi(R^{\mathsf{d}}\downarrow_p^{l^{\mathsf{c}}}) = \big\{\{b\}, \{b.f, b.g\}\big\}$$

$$\Pi(\rho_{rstc}(R^{\mathsf{u}}\downarrow_p^{l^{\mathsf{c}}} \cup R_p)) = \big\{\{b\}, \{a1, a2, b.f, b.g\}\big\}$$

$$\frac{\begin{cases} stmt_l \text{ is intraprocedural} \\ \sigma = \langle l, Tran, Gen \rangle \\ \diamondsuit \end{cases}}{\langle l, Tran, Gen, R, S \rangle \xrightarrow{\mathcal{I}} \langle l', Tran', Gen', R, S \rangle} \text{ INTRA}^{\mathcal{I}}$$

(a) Intraprocedural $\mathcal{LSL}^{\mathcal{I}}$ semantics.

$\cdot\downarrow_p^l$: $ARel \to ARel$ for any invocation $l{:}x{=}p(y_0, \ldots, y_k)$

s.t. $\forall R \in ARel : R\downarrow_p^l = \{\langle \alpha\downarrow_p^l, \beta\downarrow_p^l \rangle \mid \langle \alpha, \beta \rangle \in R \wedge \alpha, \beta \in AP_l^{\downarrow}\}$

$\rho_{rstc}(\cdot) : 2^{AP \times AP} \to ARel$

s.t. $\forall B \in 2^{AP \times AP} : \rho_{rstc}(B)$ is the reflexivity, symmetry, transitivity, and right-regularity closure [36] of B. (Thus $B \subseteq \rho_{rstc}(B) \wedge \rho_{rstc}(B) \in ARel$)

(b) Helper functions for $\mathcal{LSL}^{\mathcal{I}}$ method call semantics.

$$\frac{\begin{cases} l^{\mathsf{c}}\text{: } x{=}p(y_0, \ldots, y_k) \\ \text{the declaration of } p \text{ is ``} t\ p(t_0\ h_0, \ldots, t_k\ h_k)\{body_p\}\text{''} \\ \spadesuit \\ R^{\mathsf{d}} = \rho_{rstc}(R^{\mathsf{u}}\downarrow_p^{l^{\mathsf{c}}} \cup R_p) \\ S^{\mathsf{d}} = S^{\mathsf{u}}.\langle l^{\mathsf{c}}, Tran^{\mathsf{c}}, Gen^{\mathsf{c}}, R^{\mathsf{u}} \rangle \end{cases}}{\langle l^{\mathsf{c}}, Tran^{\mathsf{c}}, Gen^{\mathsf{c}}, R^{\mathsf{u}}, S^{\mathsf{u}} \rangle \xrightarrow{\mathcal{I}} \langle l^{\mathsf{e}}, Tran^{\mathsf{e}}, Gen^{\mathsf{e}}, R^{\mathsf{d}}, S^{\mathsf{d}} \rangle} \text{ CALL}^{\mathcal{I}}$$

(c) $\mathcal{LSL}^{\mathcal{I}}$ method call semantics.

$$\frac{\begin{cases} l^{\mathsf{x}}\text{: exit} \\ S^{\mathsf{d}} = S^{\mathsf{u}}.\langle l^{\mathsf{c}}, Tran^{\mathsf{c}}, Gen^{\mathsf{c}}, R^{\mathsf{u}} \rangle \\ l^{\mathsf{c}}\text{: } x{=}p(y_0, \ldots, y_k) \\ \text{the declaration of } p \text{ is ``} t\ p(t_0\ h_0, \ldots, t_k\ h_k)\{body_p\}\text{''} \\ \spadesuit \\ \clubsuit \end{cases}}{\langle l^{\mathsf{x}}, Tran^{\mathsf{x}}, Gen^{\mathsf{x}}, R^{\mathsf{d}}, S^{\mathsf{d}} \rangle \xrightarrow{\mathcal{I}} \langle l^{\mathsf{r}}, Tran^{\mathsf{r}}, Gen^{\mathsf{r}}, R^{\mathsf{u}}, S^{\mathsf{u}} \rangle} \text{ RETURN}^{\mathcal{I}}$$

(d) $\mathcal{LSL}^{\mathcal{I}}$ method return semantics.

Fig. 7.25.: $\mathcal{LSL}^{\mathcal{I}}$ semantics.

The return rule RETURN$^{\mathcal{I}}$ pops the pending call state from the stack and merge the call state with callee's return state in the same way as the $\mathcal{LSL}$ semantics.

Before continuing the proof, an over-approximating ordering $\cdot \sqsubseteq \cdot$ among partitions is defined below:

$$\begin{aligned}&\text{over-approximating ordering} \sqsubseteq\ \in 2^{APart \times APart}\\&\quad \text{s.t. } \forall \pi_1, \pi_2 \in APart : \pi_1 \sqsubseteq \pi_2 \text{ if and only if } \forall A_1 \in \pi_1 : \exists A_2 \in \pi_2 : A_1 \subseteq A_2\end{aligned}$$

For any two partitions π_1 and π_2, $\pi_1 \sqsubseteq \pi_2$ if and only if π_2 is coarser than π_1, i.e. $\forall A_1 \in \pi_1 : \exists A_2 \in \pi_2 : A_1 \subseteq A_2$. Furthermore, a coarsening function $\lceil \cdot \rceil^{\pi_2}$ between two partitions $\pi_1 \sqsubseteq \pi_2$ can be defined as follows:

$$\begin{aligned}&\text{coarsening function } \lceil \cdot \rceil^{\pi_2} : \pi_1 \to \pi_2 \text{ for any } \pi_1, \pi_2 \in APart \text{ s.t. } \pi_1 \sqsubseteq \pi_2\\&\quad \text{s.t. } \forall A_1 \in \pi_1 : A_1 \subseteq \lceil A_1 \rceil^{\pi_2} \in \pi_2\end{aligned}$$

Next Proposition 7.3.3 indicates that intraprocedural transitions (rule INTRA) preserve the over-approximating relation between the heap H and certain alias relation R subsuming the alias relation R_q inferred by CLIPPER from bottom up, i.e. $R_q \subseteq R$.

Proposition 7.3.3 *Given alias relations $\overline{R}$ derived from $[\![prog]\!]^{\mathbb{R}}$ of certain program prog, at any transition of rule INTRA$^{\mathcal{I}}$ in Fig. 7.25a where $stmt_l$ is in method q, the following condition*

$$R_q \subseteq R \wedge H = image(Tran) \cup Gen \sqsubseteq \Pi(R)$$

implies that $\forall \langle l', tran, gen \rangle \in succ(\langle l, Tran, Gen \rangle)$*:*

$$\forall o \in H : tran(o) \subseteq \lceil o \rceil^{\Pi(R)} \tag{7.1}$$

$$gen(\sigma) \sqsubseteq \Pi(R) \tag{7.2}$$

Proof A case-by-case analysis of *stmt*:

1. Case "l: $x = null$" – Condition 7.1 holds because no new access path is added to any object $o \in H$. Condition 7.2 holds because $gen(\sigma)$ is $\emptyset$.

2. Case "l: $x = new\ t$" – Condition 7.1 holds because $tran$ is an identity function. Condition 7.2 holds because $\{x\} \subseteq [x]_{\Pi(R)}$.

3. Case "l: $x = y$" (or "l: return z" which is handled as "l: $ret = z$")

   For any object $o \in H$ and for any $\delta \in \Delta$ s.t. $y.\delta \in o$:

   $$x.\delta \in [y.\delta]_{\Pi(R)} \quad // \ [\![l{:}x = y]\!]^{\mathbb{R}}$$

   It follows that

   $$\{x.\delta \mid y.\delta \in o\} \subseteq [y.\delta]_{\Pi(R)} = \lceil o \rceil^{\Pi(R)}$$

   Thus

   $$tran(o) = o \cup \{x.\delta \mid y.\delta \in o\} \subseteq \lceil o \rceil^{\Pi(R)}$$

   and Condition 7.1 holds.

   Condition 7.2 holds because $gen(\sigma)$ is $\emptyset$.

4. Case "l: $x = y.f$"

   For any object $o \in H$ and for any $\delta \in \Delta$ s.t. $y.f.\delta \in o$:

   $$x.\delta \in [y.f.\delta]_{\Pi(R)} \quad // \ [\![l{:}x = y.f]\!]^{\mathbb{R}}$$

   It follows that

   $$\{x.\delta \mid y.f.\delta \in o\} \subseteq [y.f.\delta]_{\Pi(R)} = \lceil o \rceil^{\Pi(R)}$$

   Thus

   $$tran(o) = o \cup \{x.\delta \mid y.f.\delta \in o\} \subseteq \lceil o \rceil^{\Pi(R)}$$

and Condition 7.1 holds.

Condition 7.2 holds because $gen(\sigma)$ is $\emptyset$.

5. Case "l: $y.f = null$" Condition 7.1 holds because no new access path is added to any object $o \in H$. Condition 7.2 holds because $gen(\sigma)$ is $\emptyset$.

6. Case "l: $y.f = x$"

$$\begin{cases} \quad [y]_H \subseteq [y]_{\Pi(R)} \quad \text{// precondition} \\ \Rightarrow [y]_H.f \subseteq [y.f]_{\Pi(R)} \quad \text{// right-regularity} \\ \qquad\qquad = [x]_{\Pi(R)} \qquad\quad \text{// } [\![l{:}y.f = x]\!]^{\mathbb{R}} \\ [x]_H \subseteq [x]_{\Pi(R)} \quad \text{// precondition} \end{cases}$$

$$\Rightarrow [y]_H.f \cup [x]_H \subseteq [x]_{\Pi(R)}$$

$$\Rightarrow \rho_c([y]_H.f \cup [x]_H) \subseteq [x]_{\Pi(R)} \quad \text{// Corollary 7.3.2}$$

$$\Rightarrow \forall \delta \in \Delta : \rho_c([y]_H.f.\delta \cup [x]_H) \subseteq [x.\delta]_{\Pi(R)} \quad \text{// right-regularity}$$

$$\Rightarrow \forall o \in H : o \ \cup \bigcup \{\rho_c([y]_H.f \cup [x]_H).\delta \mid x.\delta \in o\} \subseteq \lceil o \rceil^{\Pi(R)}$$

Thus Condition 7.1 holds.

Condition 7.2 holds because $gen(\sigma)$ is $\emptyset$.

7. Case "l: goto l'" or "l: if b l_t l_f" Condition 7.1 holds because $tran$ is an identity function. Condition 7.2 holds because $gen(\sigma)$ is $\emptyset$.

■

Next Proposition 7.3.4 indicates that call transitions (rule CALL$^{\mathcal{I}}$) preserve the over-approximating relation between the caller heap H^{c} and certain caller alias relation R^{u} subsuming the caller alias relation R_q inferred by CLIPPER from bottom up, i.e. $R_q \subseteq R^{\mathrm{u}}$.

Proposition 7.3.4 *Given alias relations $\overline{R}$ derived from $[\![prog]\!]^{\mathbb{R}}$ of program prog, at any transition of rule CALL$^{\mathcal{I}}$ in Fig. 7.25c where l^c: x=$p(y_0, \ldots, y_k)$ is in method q, the following condition*

$$R_q \subseteq R^u \wedge H^c = image(Tran^c) \cup Gen^c \sqsubseteq \Pi(R^u)$$

implies

$$\forall y_i.\delta_i \in AP : [y_i.\delta_i]_{H^c} \textit{ exists} \implies [y_i.\delta_i]_{H^c}{\downarrow} \subseteq [y_i.\delta_i{\downarrow}_p^{l^c}]_{\Pi(R^d)} \tag{7.3}$$

$$H^e \sqsubseteq \Pi(R^d) \tag{7.4}$$

Proof Because

$$\forall o^c \in H^{passed} : \forall y_i.\delta_i, y_j.\delta_j \in o^c : \langle y_i.\delta_1, y_j.\delta_2\rangle \in R^u \text{ // precondition}$$
$$\Rightarrow \forall o^c \in H^{passed} : \forall y_i.\delta_i, y_j.\delta_j \in o^c : \langle h_i.\delta_1, h_j.\delta_2\rangle \in R^d \text{ // } R^d = \rho_{rstc}(R^u{\downarrow}_p^{l^c} \cup R_p)$$
$$\Rightarrow \forall y_i.\delta_i \in AP : [y_i.\delta_i]_{H^c} \text{ exists} \implies [y_i.\delta_i]_{H^c}{\downarrow} \subseteq [y_i.\delta_i{\downarrow}_p^{l^c}]_{\Pi(R^d)}$$

condition 7.3 holds.

Condition 7.4 follows from condition 7.3. ■

Lemma 7.3.5 *Given alias relations $\overline{R}$ derived from $[\![prog]\!]^{\mathbb{R}}$ of program prog, at any transition of rule CALL$^{\mathcal{I}}$ in Fig. 7.25c where l^c: x=$p(y_0, \ldots, y_k)$ is in method q and $R_q \subseteq R^u$, the following implication holds*

$$\forall \alpha, \beta \in AP_p^{\uparrow} : \langle \alpha, \beta\rangle \in R^d \implies \langle \alpha{\uparrow}_p^{l^c}, \beta{\uparrow}_p^{l^c}\rangle \in R^u$$

Proof For any $\alpha, \beta \in AP_p^{\uparrow}$ such that $\langle \alpha, \beta \rangle \in R^{\mathtt{d}}$, there exists a sequence $(\langle \alpha_i, \alpha_i' \rangle)_{i=1}^{k}$ such that

$$\wedge \begin{cases} \alpha_1 = \alpha & (a) \\ \alpha_k' = \beta & (b) \\ \langle \alpha_i, \alpha_i' \rangle \in R_p \text{ for } 1 \leq i \leq k & (c) \\ \langle \alpha_i', \alpha_{i+1} \rangle \in R^{\mathtt{u}} \downarrow_p^{l^{\mathtt{c}}} \text{ for } 1 \leq i \leq k-1 & (d) \end{cases}$$

It follows that $\forall i \in [1, k] : \alpha_i, \alpha_i' \in AP_p^{\uparrow}$ because of (d) and hence there exists another sequence $(\langle \alpha_i \uparrow_p^{l^{\mathtt{c}}}, \alpha_i' \uparrow_p^{l^{\mathtt{c}}} \rangle)_{i=1}^{k}$ such that

$$\wedge \begin{cases} \langle \alpha_i \uparrow_p^{l^{\mathtt{c}}}, \alpha_i' \uparrow_p^{l^{\mathtt{c}}} \rangle \in R_q \subseteq R^{\mathtt{u}} \text{ for } 1 \leq i \leq k & \text{// condition (c) and definition of } \overline{R} \\ \langle \alpha_i' \uparrow_p^{l^{\mathtt{c}}}, \alpha_{i+1} \uparrow_p^{l^{\mathtt{c}}} \rangle \in R^{\mathtt{u}} \text{ for } 1 \leq i \leq k-1 & \text{// condition (d)} \end{cases}$$

$$\Rightarrow \langle \alpha_1 \uparrow_p^{l^{\mathtt{c}}}, \alpha_k' \uparrow_p^{l^{\mathtt{c}}} \rangle \in R^{\mathtt{u}} \text{ // by transitivity}$$

$$\Rightarrow \langle \alpha \uparrow_p^{l^{\mathtt{c}}}, \beta \uparrow_p^{l^{\mathtt{c}}} \rangle \in R^{\mathtt{u}} \text{ // condition (a) and (b)}$$

■

The following Proposition 7.3.6 and Corollary 7.3.7 indicate that return transitions (rule RETURN$^{\mathcal{I}}$) preserve the over-approximating relation between the caller heap $H^{\mathtt{c}}$ and certain caller alias relation $R^{\mathtt{u}}$ subsuming the caller alias relation R_q inferred by CLIPPER from bottom up (i.e. $R_q \subseteq R^{\mathtt{u}}$) if the callee object transformer $Tran^{\mathtt{x}}$ preserves the over-approximating relation between the callee heap $H^{\mathtt{e}}$ and the callee alias relation $R^{\mathtt{d}}$ (condition (c)) and the callee generated object set $Gen^{\mathtt{x}}$ is over-approximated by the callee alias relation $R^{\mathtt{d}}$ (condition (d)).

Proposition 7.3.6 *Given alias relations $\overline{R}$ derived from $[\![prog]\!]^{\mathbb{R}}$ of program prog, at any transition of rule RETURN$^{\mathcal{I}}$ in Fig. 7.25d where l^c: $x{=}p(y_0,\ldots,y_k)$ is in method q, the following condition*

$$\wedge\begin{cases} H^c = image(Tran^c) \cup Gen^c \sqsubseteq \Pi(R^u) & (a) \\ R_q \subseteq R^u & (b) \\ \forall o^e \in H^e : Tran^x(o^e) \subseteq \lceil o^e \rceil^{\Pi(R^d)} & (c) \\ Gen^x \sqsubseteq \Pi(R^d) & (d) \end{cases}$$

implies

$$\forall o^x \in H^x : \exists A^u \in \Pi(R^u) : o^x{\uparrow} \subseteq A^u \tag{7.5}$$

$$\forall i \in [0,k] \text{ and } \delta \in \Delta : [y_i.\delta_i]_{H^c} \text{ exists } \Longrightarrow Tran^x([h_i.\delta_i]_{H^e}){\uparrow} \subseteq [y_i.\delta_i]_{\Pi(R^u)} \tag{7.6}$$

Proof For any $o^{\mathrm{x}} \in H^{\mathrm{x}}$, $\cdot\!\uparrow$ maps the following two kinds of access paths to caller

$$\begin{cases} [ret_p]_{H^{\mathrm{x}}}.\delta' \subseteq o^{\mathrm{x}} & \text{for } o^{\mathrm{x}ret} = [ret_p]_{H^{\mathrm{x}}} \\ Tran^{\mathrm{x}}([h_i.\delta]_{H^{\mathrm{e}}}).\delta' \subseteq o^{\mathrm{x}} & \text{for } [h_i.\delta]_{H^{\mathrm{e}}} \in H^{params} \cup H^{cpl} \end{cases}$$

It follows that

$$\begin{cases} H^{\mathsf{e}} \sqsubseteq \Pi(R^{\mathsf{d}}) \quad \text{// Proposition 7.3.4} \\ H^{\mathsf{x}} = image(Tran^{\mathsf{x}}) \cup Gen^{\mathsf{x}} \sqsubseteq \Pi(R^{\mathsf{d}}) \quad \text{// condition (c) and (d)} \end{cases}$$

$$\Rightarrow \forall i \in [0,k] \text{ and } \delta \in \Delta :$$

$$h_i.\delta \in \lceil [h_i.\delta]_{H^{\mathsf{e}}} \rceil^{\Pi(R^{\mathsf{d}})} = \lceil Tran^{\mathsf{x}}([h_i.\delta]_{H^{\mathsf{e}}}) \rceil^{\Pi(R^{\mathsf{d}})} \quad \text{// condition (c)}$$

$$\Rightarrow \forall i \in [0,k] \text{ and } \delta, \delta' \in \Delta :$$

$$Tran^{\mathsf{x}}([h_i.\delta]_{H^{\mathsf{e}}}).\delta' \subseteq o^{\mathsf{x}} \Rightarrow h_i.\delta.\delta' \in \lceil o^{\mathsf{x}} \rceil^{\Pi(R^{\mathsf{d}})} \quad \text{// right-regularity of } R^{\mathsf{d}}$$

$$\Rightarrow \forall \alpha \in \{h_i.\delta.\delta' \mid Tran^{\mathsf{x}}([h_i.\delta]_{H^{\mathsf{e}}}).\delta' \subseteq o^{\mathsf{x}}\} : \alpha \in \lceil o^{\mathsf{x}} \rceil^{\Pi(R^{\mathsf{d}})}$$

$$\Rightarrow \forall \alpha \in \{ret_p.\delta' \mid ret_p.\delta' \in o^{\mathsf{x}}\} \cup \{h_i.\delta.\delta' \mid Tran^{\mathsf{x}}([h_i.\delta]_{H^{\mathsf{e}}}).\delta' \subseteq o^{\mathsf{x}}\} : \alpha \in \lceil o^{\mathsf{x}} \rceil^{\Pi(R^{\mathsf{d}})}$$

$$\Rightarrow \forall \alpha, \beta \in \{ret_p.\delta' \mid ret_p.\delta' \in o^{\mathsf{x}}\} \cup \{h_i.\delta.\delta' \mid Tran^{\mathsf{x}}([h_i.\delta]_{H^{\mathsf{e}}}).\delta' \subseteq o^{\mathsf{x}}\} : \langle \alpha, \beta \rangle \in R^{\mathsf{d}}$$

$$\Rightarrow \forall \alpha, \beta \in \{x.\delta' \mid ret_p.\delta' \in o^{\mathsf{x}}\} \cup \{y_i.\delta.\delta' \mid Tran^{\mathsf{x}}([h_i.\delta]_{H^{\mathsf{e}}}).\delta' \subseteq o^{\mathsf{x}}\} :$$

$$\langle \alpha, \beta \rangle \in R^{\mathsf{u}} \text{ // condition (b) and Lemma 7.3.5}$$

$$\Rightarrow \forall \alpha, \beta \in \{x.\delta' \mid ret_p.\delta' \in o^{\mathsf{x}}\} \cup \bigcup \{[y_i.\delta]_{H^{\mathsf{c}}}.\delta' \mid Tran^{\mathsf{x}}([h_i.\delta]_{H^{\mathsf{e}}}).\delta' \subseteq o^{\mathsf{x}}\} :$$

$$\langle \alpha, \beta \rangle \in R^{\mathsf{u}} \text{ // condition (a) and right-regularity of } R^{\mathsf{u}}$$

$$\Rightarrow \exists A^{\mathsf{u}} \in \Pi(R^{\mathsf{u}}) : o^{\mathsf{x}}\!\uparrow \subseteq A^{\mathsf{u}} \text{ // by definition of } \cdot\!\uparrow$$

Hence condition 7.5 holds.

Given $[y_i.\delta_i]_{H^{\mathsf{c}}}$ exists, it follows that

$$\begin{cases} \forall \alpha, \beta \in \{x.\delta' \mid ret_p.\delta' \in Tran^{\mathsf{x}}([h_i.\delta_i]_{H^{\mathsf{e}}})\} \cup \\ \quad \bigcup \{[y_j.\delta_j]_{H^{\mathsf{c}}}.\delta' \mid Tran^{\mathsf{x}}([h_j.\delta_j]_{H^{\mathsf{e}}}).\delta' \subseteq Tran^{\mathsf{x}}([h_i.\delta_i]_{H^{\mathsf{e}}})\} : \\ \quad \langle \alpha, \beta \rangle \in R^{\mathsf{u}} \text{ // derived above} \\ y_i.\delta_i \in \bigcup \{[y_j.\delta_j]_{H^{\mathsf{c}}}.\delta' \mid Tran^{\mathsf{x}}([h_j.\delta_j]_{H^{\mathsf{e}}}).\delta' \subseteq Tran^{\mathsf{x}}([h_i.\delta_i]_{H^{\mathsf{e}}})\} \end{cases}$$

$$\Rightarrow Tran^{\mathsf{x}}([h_i.\delta_i]_{H^{\mathsf{e}}})\!\uparrow \subseteq [y_i.\delta_i]_{\Pi(R^{\mathsf{u}})} \text{ // by definition of } \cdot\!\uparrow$$

Hence condition 7.6 holds.

■

Corollary 7.3.7 *Given alias relations $\overline{R}$ derived from $[\![prog]\!]^{\mathbb{R}}$ of program prog, at any transition of rule $RETURN^{\mathcal{I}}$ in Fig. 7.25d where l^{c}: $x{=}p(y_0, \ldots, y_k)$ is in method q, the following condition*

$$\wedge \begin{cases} H^{\mathsf{c}} = image(Tran^{\mathsf{c}}) \cup Gen^{\mathsf{c}} \sqsubseteq \Pi(R^{u}) \\ R_q \subseteq R^{u} \\ \forall o^{e} \in H^{e} : Tran^{x}(o^{e}) \subseteq \lceil o^{e} \rceil^{\Pi(R^{d})} \\ Gen^{x} \sqsubseteq \Pi(R^{d}) \end{cases}$$

implies

$$\forall o^{\mathsf{c}} \in H^{\mathsf{c}} : tran(o^{\mathsf{c}}) \subseteq \lceil o^{\mathsf{c}} \rceil^{\Pi(R^{u})} \tag{7.7}$$

$$gen \sqsubseteq \Pi(R^{u}) \tag{7.8}$$

Proof To prove condition 7.7, two cases need to be considered.

1. First consider the case where $o^{\mathsf{c}} \in H^{\mathsf{c}} \setminus H^{passed}$. It follows that

$$tran(o^{\mathsf{c}}) = o^{\mathsf{c}} \subseteq \lceil o^{\mathsf{c}} \rceil^{\Pi(R^{\mathsf{u}})}$$

2. Next consider the case where $o^{\mathsf{c}} \in H^{passed}$. Then there exists $y_i.\delta_i \in o^{\mathsf{c}}$. It follows that

$$tran(o^{\mathsf{c}}) = Tran^{\mathsf{x}}([h_i.\delta_i]_{H^{\mathsf{e}}}){\uparrow}\subseteq [y_i.\delta_i]_{\Pi(R^{\mathsf{u}})} = \lceil o^{\mathsf{c}} \rceil^{\Pi(R^{\mathsf{u}})}$$

Hence condition 7.7 holds.

Condition 7.8 follows from the definition of *gen* and condition 7.5. ■

The same-level realizable paths ($SLRP$s) are defined as in [22]. Note: An $SLRP$ starts at an entry state $\langle l^{\mathsf{e}}, Tran^{\mathsf{e}}, Gen^{\mathsf{e}}, R, S\rangle$.

path $\rho, \hat{\sigma}_0 \xrightarrow{\mathcal{I}} \hat{\sigma}_1 \xrightarrow{\mathcal{I}} \cdots \xrightarrow{\mathcal{I}} \hat{\sigma}_k \in \mathit{Path} = (\xrightarrow{\mathcal{I}})^*$
empty path $\hat{\sigma} \in \mathit{Path}^0 \subset \mathit{Path}$
path length $|\cdot| : \mathit{Path} \to \mathbb{N}$

$$\text{s.t. } \forall \rho \in \mathit{Path} : |\rho| = \begin{cases} 0 & \text{if } \rho \in \mathit{Path}^0 \\ |\hat{\sigma}_0 \xrightarrow{\mathcal{I}} \cdots \xrightarrow{\mathcal{I}} \hat{\sigma}_k| + 1 & \text{if } \rho = \hat{\sigma}_0 \xrightarrow{\mathcal{I}} \cdots \xrightarrow{\mathcal{I}} \hat{\sigma}_k \xrightarrow{\mathcal{I}} \hat{\sigma}_{k+1} \end{cases}$$

same-level realizable path $s \in \mathit{SLRP} \subset \mathit{Path}$
s.t. $s \in \mathit{SLRP}$ if

$$\bigvee \begin{cases} s = \langle l^{\mathtt{e}}, \mathit{Tran}^{\mathtt{e}}, \mathit{Gen}^{\mathtt{e}}, R, S \rangle \quad \text{// } s \text{ is empty} \\ s = \langle l^{\mathtt{e}}, \mathit{Tran}^{\mathtt{e}}, \mathit{Gen}^{\mathtt{e}}, R, S \rangle \xrightarrow{\mathcal{I}} \cdots \xrightarrow{\mathcal{I}} \langle l, \mathit{Tran}, \mathit{Gen}, R, S \rangle \\ \quad \xrightarrow{\mathcal{I}} \langle l', \mathit{Tran}', \mathit{Gen}', R, S \rangle \wedge \\ \quad \langle l^{\mathtt{e}}, \mathit{Tran}^{\mathtt{e}}, \mathit{Gen}^{\mathtt{e}}, R, S \rangle \xrightarrow{\mathcal{I}} \cdots \xrightarrow{\mathcal{I}} \langle l, \mathit{Tran}, \mathit{Gen}, R, S \rangle \in \mathit{SLRP} \wedge \\ \quad \mathit{stmt}_l \text{ is intraprocedural} \\ s = \langle l^{\mathtt{e}}_q, \mathit{Tran}^{\mathtt{e}}_q, \mathit{Gen}^{\mathtt{e}}_q, R^{\mathtt{u}}, S^{\mathtt{u}} \rangle \xrightarrow{\mathcal{I}} \cdots \xrightarrow{\mathcal{I}} \langle l^{\mathtt{c}}_q, \mathit{Tran}^{\mathtt{c}}_q, \mathit{Gen}^{\mathtt{c}}_q, R^{\mathtt{u}}, S^{\mathtt{u}} \rangle \\ \quad \xrightarrow{\mathcal{I}} \langle l^{\mathtt{e}}_p, \mathit{Tran}^{\mathtt{e}}_p, \mathit{Gen}^{\mathtt{e}}_p, R^{\mathtt{d}}, S^{\mathtt{d}} \rangle \xrightarrow{\mathcal{I}} \cdots \xrightarrow{\mathcal{I}} \langle l^{\mathtt{x}}_p, \mathit{Tran}^{\mathtt{x}}_p, \mathit{Gen}^{\mathtt{x}}_p, R^{\mathtt{d}}, S^{\mathtt{d}} \rangle \\ \quad \xrightarrow{\mathcal{I}} \langle l^{\mathtt{r}}_q, \mathit{Tran}^{\mathtt{r}}_q, \mathit{Gen}^{\mathtt{r}}_q, R^{\mathtt{u}}, S^{\mathtt{u}} \rangle \wedge \\ \quad \langle l^{\mathtt{e}}_q, \mathit{Tran}^{\mathtt{e}}_q, \mathit{Gen}^{\mathtt{e}}_q, R^{\mathtt{u}}, S^{\mathtt{u}} \rangle \xrightarrow{\mathcal{I}} \cdots \xrightarrow{\mathcal{I}} \langle l^{\mathtt{c}}_q, \mathit{Tran}^{\mathtt{c}}_q, \mathit{Gen}^{\mathtt{c}}_q, R^{\mathtt{u}}, S^{\mathtt{u}} \rangle \in \mathit{SLRP} \wedge \\ \quad l^{\mathtt{c}}_q\text{: } x{=}p(y_0, \ldots, y_k) \wedge \\ \quad \langle l^{\mathtt{e}}_p, \mathit{Tran}^{\mathtt{e}}_p, \mathit{Gen}^{\mathtt{e}}_p, R^{\mathtt{d}}, S^{\mathtt{d}} \rangle \xrightarrow{\mathcal{I}} \cdots \xrightarrow{\mathcal{I}} \langle l^{\mathtt{x}}_p, \mathit{Tran}^{\mathtt{x}}_p, \mathit{Gen}^{\mathtt{x}}_p, R^{\mathtt{d}}, S^{\mathtt{d}} \rangle \in \mathit{SLRP} \wedge \\ \quad l^{\mathtt{x}}_p\text{: exit} \end{cases}$$

Fig. 7.26.: Definiton of same-level realizable path (SLRP).

Example 18 *In Fig. 7.11 and 7.12 for example, the following traces are SLRPs*

$$\hat{\sigma}_{13} \xrightarrow{\mathcal{I}} \cdots \xrightarrow{\mathcal{I}} \hat{\sigma}_{15}$$
$$\hat{\sigma}_{13} \xrightarrow{\mathcal{I}} \cdots \xrightarrow{\mathcal{I}} \hat{\sigma}_{16} \xrightarrow{\mathcal{I}} \hat{\sigma}_{19} \xrightarrow{\mathcal{I}} \cdots \xrightarrow{\mathcal{I}} \hat{\sigma}_{23} \xrightarrow{\mathcal{I}} {}_{17}$$
$$\hat{\sigma}_{19} \xrightarrow{\mathcal{I}} \cdots \xrightarrow{\mathcal{I}} \hat{\sigma}_{22}$$

where $\hat{\sigma}_i = \langle l_i, Tran_i, Gen_i, R, S\rangle$ *i.e. extending* $\sigma_i = \langle l_i, Tran_i, Gen_i\rangle$ *with proper* R *and* S*.*

In Fig. 7.13 and 7.14 for example, the following traces are SLRPs

$$\hat{\sigma}_{24} \xrightarrow{\mathcal{I}} \cdots \xrightarrow{\mathcal{I}} \hat{\sigma}_{26}$$
$$\hat{\sigma}_{24} \xrightarrow{\mathcal{I}} \cdots \xrightarrow{\mathcal{I}} \hat{\sigma}_{26} \xrightarrow{\mathcal{I}} \hat{\sigma}_{28} \xrightarrow{\mathcal{I}} \cdots \xrightarrow{\mathcal{I}} \hat{\sigma}_{32} \xrightarrow{\mathcal{I}} {}_{27}$$
$$\hat{\sigma}_{28} \xrightarrow{\mathcal{I}} \cdots \xrightarrow{\mathcal{I}} \hat{\sigma}_{31}$$

Next Proposition 7.3.8 indicates that *SLRP*s preserve the over-approximating relation between the heap H^{e} and certain alias relation R subsuming the alias relation R_q inferred by CLIPPER from bottom up (i.e. $R_q \subseteq R$).

Proposition 7.3.8 *Given alias relations* $\overline{R}$ *derived from* $[\![prog]\!]^{\mathbb{R}}$ *of program prog and any SLRP* $s = \langle l^{e}, Tran^{e}, Gen^{e}, R, S\rangle \xrightarrow{\mathcal{I}} \cdots \xrightarrow{\mathcal{I}} \langle l, Tran, Gen, R, S\rangle$ *from an entry state of any method* q*, the following condition*

$$\wedge \begin{cases} H^{e} = dom(Tran^{e}) \sqsubseteq \Pi(R) & (a) \\ \forall o^{e} \in H^{e} : Tran^{e}(o^{e}) = o^{e} & (b) \\ Gen^{e} = \emptyset & (c) \\ R_q \subseteq R & (d) \end{cases}$$

implies

$$\forall o^{\mathsf{e}} \in H^{\mathsf{e}} : Tran(o^{\mathsf{e}}) \subseteq \lceil o^{\mathsf{e}} \rceil^{\Pi(R)} \tag{7.9}$$

$$Gen \sqsubseteq \Pi(R) \tag{7.10}$$

Proof We proceed by well-founded induction on s.

1. First consider the base case where s is empty, i.e. $s = \langle l^{\mathsf{e}}, Tran^{\mathsf{e}}, Gen^{\mathsf{e}}, R, S \rangle$

   Because

   $$\forall o^{\mathsf{e}} \in H^{\mathsf{e}} : Tran(o^{\mathsf{e}}) = Tran^{\mathsf{e}}(o^{\mathsf{e}}) = o^{\mathsf{e}} \subseteq \lceil o^{\mathsf{e}} \rceil^{\Pi(R)}$$

   condition 7.9 holds.

   Condition 7.10 holds because $Gen = Gen^{\mathsf{e}} = \emptyset$.

2. Next consider the inductive case where

   $$\begin{aligned} &\forall s' \in SLRP : \\ &\quad \wedge \begin{cases} |s'| < |s| \\ s' \text{ starts from an entry state satisfying conditions (a) and (b)} \end{cases} \\ &\quad \Longrightarrow \text{ conditions 7.9 and 7.10 hold on } s' \end{aligned}$$

   Two cases need to be considered:

   (a) First consider the case

   $$\begin{aligned} s = \langle l^{\mathsf{e}}, Tran^{\mathsf{e}}, Gen^{\mathsf{e}}, R, S \rangle &\xrightarrow{\mathcal{I}} \cdots \xrightarrow{\mathcal{I}} \langle l, Tran, Gen, R, S \rangle \\ &\xrightarrow{\mathcal{I}} \langle l', Tran', Gen', R, S \rangle \end{aligned}$$

where

$$\wedge\begin{cases}\langle l^{\mathsf{e}}, Tran^{\mathsf{e}}, Gen^{\mathsf{e}}, R, S\rangle \xrightarrow{\mathcal{I}} \cdots \xrightarrow{\mathcal{I}} \langle l, Tran, Gen, R, S\rangle \in SLRP \\ stmt_l \text{ is intraprocedural}\end{cases}$$

Because

$$\begin{aligned}
&\text{Let } \sigma = \langle l, Tran, Gen\rangle \text{ in} \\
&\forall o^{\mathsf{e}} \in H^{\mathsf{e}} : Tran(o^{\mathsf{e}}) \subseteq \lceil o^{\mathsf{e}} \rceil^{\Pi(R)} \qquad \text{// induction hypothesis} \\
\Rightarrow &\forall o^{\mathsf{e}} \in H^{\mathsf{e}} : \\
&\quad Tran'(o^{\mathsf{e}}) = tran(Tran(o^{\mathsf{e}})) \subseteq \lceil Tran(o^{\mathsf{e}}) \rceil^{\Pi(R)} = \lceil o^{\mathsf{e}} \rceil^{\Pi(R)} \\
&\qquad\qquad \text{// Proposition 7.3.3}
\end{aligned}$$

condition 7.9 holds.

Because

$$\begin{aligned}
&\text{Let } \sigma = \langle l, Tran, Gen\rangle \text{ in} \\
&\begin{cases}\forall o \in Gen : \exists A \in \Pi(R) : o \subseteq A & \text{// induction hypothesis} \\ \quad \Rightarrow \forall o \in Gen : tran(o) \subseteq \lceil o \rceil^{\Pi(R)} & \text{// Proposition 7.3.3} \\ \forall o \in gen(\sigma) : \exists A \in \Pi(R) : o \subseteq A & \text{// Proposition 7.3.3}\end{cases} \\
\Rightarrow &\forall o' \in Gen' : \exists A \in \Pi(R) : o' \subseteq A
\end{aligned}$$

condition 7.10 holds.

(b) Next consider the case

$$\begin{aligned}
s = &\langle l^{\mathsf{e}}_q, Tran^{\mathsf{e}}_q, Gen^{\mathsf{e}}_q, R^{\mathsf{u}}, S^{\mathsf{u}}\rangle \xrightarrow{\mathcal{I}} \cdots \xrightarrow{\mathcal{I}} \langle l^{\mathsf{c}}_q, Tran^{\mathsf{c}}_q, Gen^{\mathsf{c}}_q, R^{\mathsf{u}}, S^{\mathsf{u}}\rangle \\
&\xrightarrow{\mathcal{I}} \langle l^{\mathsf{e}}_p, Tran^{\mathsf{e}}_p, Gen^{\mathsf{e}}_p, R^{\mathsf{d}}, S^{\mathsf{d}}\rangle \xrightarrow{\mathcal{I}} \cdots \xrightarrow{\mathcal{I}} \langle l^{\mathsf{x}}_p, Tran^{\mathsf{x}}_p, Gen^{\mathsf{x}}_p, R^{\mathsf{d}}, S^{\mathsf{d}}\rangle \\
&\xrightarrow{\mathcal{I}} \langle l^{\mathsf{r}}_q, Tran^{\mathsf{r}}_q, Gen^{\mathsf{r}}_q, R^{\mathsf{u}}, S^{\mathsf{u}}\rangle
\end{aligned}$$

where

$$\wedge \begin{cases} \langle l_q^{\mathtt{e}}, Tran_q^{\mathtt{e}}, Gen_q^{\mathtt{e}}, R^{\mathtt{u}}, S^{\mathtt{u}}\rangle \xrightarrow{\mathcal{I}} \cdots \xrightarrow{\mathcal{I}} \langle l_q^{\mathtt{c}}, Tran_q^{\mathtt{c}}, Gen_q^{\mathtt{c}}, R^{\mathtt{u}}, S^{\mathtt{u}}\rangle \in SLRP \\ l_q^{\mathtt{c}}\text{: } x{=}p(y_0, \ldots, y_k) \\ \langle l_p^{\mathtt{e}}, Tran_p^{\mathtt{e}}, Gen_p^{\mathtt{e}}, R^{\mathtt{d}}, S^{\mathtt{d}}\rangle \xrightarrow{\mathcal{I}} \cdots \xrightarrow{\mathcal{I}} \langle l_p^{\mathtt{x}}, Tran_p^{\mathtt{x}}, Gen_p^{\mathtt{x}}, R^{\mathtt{d}}, S^{\mathtt{d}}\rangle \in SLRP \\ l_p^{\mathtt{x}}\text{: exit} \end{cases}$$

It follows that

$$\begin{cases} \begin{cases} \forall o_q^{\mathtt{e}} \in H_q^{\mathtt{e}} : Tran_q^{\mathtt{c}}(o_q^{\mathtt{e}}) \subseteq \lceil o_q^{\mathtt{e}} \rceil^{\Pi(R^{\mathtt{u}})} & \text{// induction hypothesis} \\ \forall o_q^{\mathtt{c}} \in Gen_q^{\mathtt{c}} : \exists A^{\mathtt{u}} \in \Pi(R^{\mathtt{u}}) : o_q^{\mathtt{c}} \subseteq A^{\mathtt{u}} & \text{// induction hypothesis} \end{cases} \\ \quad \Rightarrow H_q^{\mathtt{c}} = image(Tran_q^{\mathtt{c}}) \cup Gen_q^{\mathtt{c}} \sqsubseteq \Pi(R^{\mathtt{u}}) \\ \quad \Rightarrow H_p^{\mathtt{e}} \sqsubseteq R^{\mathtt{d}} \quad \text{// Proposition 7.3.4} \\ R_p \subseteq R^{\mathtt{d}} \quad \text{// by definition of } R^{\mathtt{d}} \end{cases}$$

$$\Rightarrow \begin{cases} \forall o_p^{\mathtt{e}} \in H_p^{\mathtt{e}} : Tran_p^{\mathtt{x}}(o_p^{\mathtt{e}}) \subseteq \lceil o_p^{\mathtt{e}} \rceil^{\Pi(R^{\mathtt{d}})} \\ \forall o_p^{\mathtt{x}} \in Gen_p^{\mathtt{x}} : \exists A^{\mathtt{d}} \in \Pi(R^{\mathtt{d}}) : o_p^{\mathtt{x}} \subseteq A^{\mathtt{d}} \end{cases} \quad \text{// induction hypothesis}$$

$$\Rightarrow \begin{cases} \forall o_q^{\mathtt{c}} \in H_q^{\mathtt{c}} : tran^{l_q^{\mathtt{c}}}(o_q^{\mathtt{c}}) \subseteq \lceil o_q^{\mathtt{c}} \rceil^{\Pi(R^{\mathtt{u}})} & (c) \\ \forall o_q^{\mathtt{r}} \in gen^{l_q^{\mathtt{c}}} : \exists A^{\mathtt{u}} \in \Pi(R^{\mathtt{u}}) : o_q^{\mathtt{r}} \subseteq A^{\mathtt{u}} & (d) \end{cases} \quad \text{// Corollary 7.3.7}$$

Because

$$\begin{aligned} &\forall o_q^{\mathtt{e}} \in H_q^{\mathtt{e}} : \\ &\quad Tran_q^{\mathtt{r}}(o_q^{\mathtt{e}}) = tran^{l_q^{\mathtt{c}}}(Tran_q^{\mathtt{c}}(o_q^{\mathtt{e}})) \subseteq \lceil Tran_q^{\mathtt{c}}(o_q^{\mathtt{e}}) \rceil^{\Pi(R^{\mathtt{u}})} = \lceil o_q^{\mathtt{e}} \rceil^{\Pi(R^{\mathtt{u}})} \end{aligned}$$

condition 7.9 holds.

Because

$$\begin{cases} \forall o_q^{\mathtt{c}} \in Gen_q^{\mathtt{c}} : \exists A^{\mathtt{u}} \in \Pi(R^{\mathtt{u}}) : o_q^{\mathtt{c}} \subseteq A^{\mathtt{u}} & \text{// induction hypothesis} \\ \quad \Rightarrow \forall o_q^{\mathtt{c}} \in Gen_q^{\mathtt{c}} : tran^{l_q^{\mathtt{c}}}(o_q^{\mathtt{c}}) \subseteq \lceil o_q^{\mathtt{c}} \rceil^{\Pi(R^{\mathtt{u}})} & \text{// condition (c)} \\ \forall o_q^{\mathtt{r}} \in gen^{l_q^{\mathtt{c}}} : \exists A^{\mathtt{u}} \in \Pi(R^{\mathtt{u}}) : o_q^{\mathtt{r}} \subseteq A^{\mathtt{u}} & \text{// condition (d)} \end{cases}$$

$$\Rightarrow \forall o_q^{\mathtt{r}} \in Gen_q^{\mathtt{r}} = gen^{l_q^{\mathtt{c}}} \cup map(tran^{l_q^{\mathtt{c}}})(Gen_q^{\mathtt{c}}) : \exists A^{\mathtt{u}} \in \Pi(R^{\mathtt{u}}) : o_q^{\mathtt{r}} \subseteq A^{\mathtt{u}}$$

condition 7.10 holds.

■

The realizable paths are defined as in [22].

realizable path $r \in RP \subset Path$

s.t. $r \in RP$ if

$$\bigvee \begin{cases} r = \langle l^0, Tran^0, Gen^0, R^0, S^0 \rangle \quad \text{// } r \text{ is empty} \\ r = \langle l^0, Tran^0, Gen^0, R^0, S^0 \rangle \xrightarrow{\mathcal{I}} \cdots \xrightarrow{\mathcal{I}} \langle l, Tran, Gen, R, S \rangle \\ \quad \xrightarrow{\mathcal{I}} \langle l', Tran', Gen', R, S \rangle \wedge \\ \quad \langle l^0, Tran^0, Gen^0, R^0, S^0 \rangle \xrightarrow{\mathcal{I}} \cdots \xrightarrow{\mathcal{I}} \langle l, Tran, Gen, R, S \rangle \in RP \wedge \\ \quad stmt_l \text{ is not } l\text{: exit} \\ r = \langle l^0, Tran^0, Gen^0, R^0, S^0 \rangle \xrightarrow{\mathcal{I}} \cdots \xrightarrow{\mathcal{I}} \langle l_q^{\mathtt{c}}, Tran_q^{\mathtt{c}}, Gen_q^{\mathtt{c}}, R^{\mathtt{u}}, S^{\mathtt{u}} \rangle \\ \quad \xrightarrow{\mathcal{I}} \langle l_p^{\mathtt{e}}, Tran_p^{\mathtt{e}}, Gen_p^{\mathtt{e}}, R^{\mathtt{d}}, S^{\mathtt{d}} \rangle \xrightarrow{\mathcal{I}} \cdots \xrightarrow{\mathcal{I}} \langle l_p^{\mathtt{x}}, Tran_p^{\mathtt{x}}, Gen_p^{\mathtt{x}}, R^{\mathtt{d}}, S^{\mathtt{d}} \rangle \\ \quad \xrightarrow{\mathcal{I}} \langle l_q^{\mathtt{r}}, Tran_q^{\mathtt{r}}, Gen_q^{\mathtt{r}}, R^{\mathtt{u}}, S^{\mathtt{u}} \rangle \wedge \\ \quad \langle l^0, Tran^0, Gen^0, R^0, S^0 \rangle \xrightarrow{\mathcal{I}} \cdots \xrightarrow{\mathcal{I}} \langle l_q^{\mathtt{c}}, Tran_q^{\mathtt{c}}, Gen_q^{\mathtt{c}}, R^{\mathtt{u}}, S^{\mathtt{u}} \rangle \in RP \wedge \\ \quad l_q^{\mathtt{c}}\text{: } x{=}p(y_0, \ldots, y_k) \wedge \\ \quad \langle l_p^{\mathtt{e}}, Tran_p^{\mathtt{e}}, Gen_p^{\mathtt{e}}, R^{\mathtt{d}}, S^{\mathtt{d}} \rangle \xrightarrow{\mathcal{I}} \cdots \xrightarrow{\mathcal{I}} \langle l_p^{\mathtt{x}}, Tran_p^{\mathtt{x}}, Gen_p^{\mathtt{x}}, R^{\mathtt{d}}, S^{\mathtt{d}} \rangle \in SLRP \wedge \\ \quad l_p^{\mathtt{x}}\text{: exit} \end{cases}$$

Fig. 7.27.: Definition of realizable path (RP).

Example 19 *In Fig. 7.11 and 7.12 for example, the following traces are RPs*

$$\hat{\sigma}_{13} \xrightarrow{\mathcal{I}} \cdots \xrightarrow{\mathcal{I}} \hat{\sigma}_{15}$$
$$\hat{\sigma}_{13} \xrightarrow{\mathcal{I}} \cdots \xrightarrow{\mathcal{I}} \hat{\sigma}_{16} \xrightarrow{\mathcal{I}} {}_{19} \xrightarrow{\mathcal{I}} {}_{20}$$
$$\hat{\sigma}_{19} \xrightarrow{\mathcal{I}} \cdots \xrightarrow{\mathcal{I}} \hat{\sigma}_{22}$$

where $\hat{\sigma}_i = \langle l_i, Tran_i, Gen_i, R, S\rangle$ *i.e. extending* $\sigma_i = \langle l_i, Tran_i, Gen_i\rangle$ *with proper* R *and* S.

In Fig. 7.13 and 7.14 for example, the following traces are RPs

$$\hat{\sigma}_{24} \xrightarrow{\mathcal{I}} \cdots \xrightarrow{\mathcal{I}} \hat{\sigma}_{26}$$
$$\hat{\sigma}_{24} \xrightarrow{\mathcal{I}} \cdots \xrightarrow{\mathcal{I}} \hat{\sigma}_{26} \xrightarrow{\mathcal{I}} \hat{\sigma}_{28} \xrightarrow{\mathcal{I}} \cdots \xrightarrow{\mathcal{I}} \hat{\sigma}_{30}$$
$$\hat{\sigma}_{28} \xrightarrow{\mathcal{I}} \cdots \xrightarrow{\mathcal{I}} \hat{\sigma}_{31}$$

Next Proposition 7.3.9 indicates that RPs preserve the over-approximating relation between the heap H^0 and certain alias relation R^0 subsuming the alias relation R_{main} inferred by Clipper from bottom up (i.e. $R_{main} \subseteq R^0$).

Proposition 7.3.9 *Given alias relations* $\overline{R}$ *derived from* $[\![prog]\!]^{\mathbb{R}}$ *of program prog and any RP* $r = \langle l^0, Tran^0, Gen^0, R^0, S^0\rangle \xrightarrow{\mathcal{I}} \cdots \xrightarrow{\mathcal{I}} \langle l, Tran, Gen, R, S\rangle$ *from an initial state of the entry method* `main` *the following condition*

$$\wedge \begin{cases} H^0 = dom(Tran^0) \sqsubseteq \Pi(R^0) & (a) \\ \forall o^0 \in H^0 : Tran^0(o^0) = o^0 & (b) \\ Gen^0 = \emptyset & (c) \\ R_{main} \subseteq R^0 & (d) \end{cases}$$

implies

$$H^{\mathsf{e}} = dom(Tran) \sqsubseteq \Pi(R) \tag{7.11}$$

$$\forall o^{\mathsf{e}} \in H^{\mathsf{e}} : Tran(o^{\mathsf{e}}) \subseteq \lceil o^{\mathsf{e}} \rceil^{\Pi(R)} \tag{7.12}$$

$$Gen \sqsubseteq \Pi(R) \tag{7.13}$$

Proof We proceed by well-founded induction on r.

1. First consider the base case where r is empty, i.e.

$$r = \langle l^0, Tran^0, Gen^0, R^0, S^0 \rangle$$

Because $H^{\mathsf{e}} = H^0 \sqsubseteq \Pi(R^0) = \Pi(R)$, condition 7.11 holds.

Because

$$\begin{aligned} &\forall o^{\mathsf{e}} \in H^{\mathsf{e}} = H^0 : \\ &\quad Tran(o^{\mathsf{e}}) = Tran^0(o^{\mathsf{e}}) = o^{\mathsf{e}} \subseteq \lceil o^{\mathsf{e}} \rceil^{\Pi(R^0)} = \lceil o^{\mathsf{e}} \rceil^{\Pi(R)} \end{aligned}$$

condition 7.12 holds.

Condition 7.13 holds because $Gen = Gen^0 = \emptyset$.

2. Next consider the inductive case where

$$\begin{aligned} &\forall r' \in RP : \\ &\quad \wedge \begin{cases} |r'| < |r| \\ r' \text{ starts from an initial state satisfying conditions (a)-(d)} \end{cases} \\ &\quad \Longrightarrow \text{ conditions 7.11, 7.12, and 7.13 hold on } r' \end{aligned}$$

Three cases need to be considered:

(a) First consider the case

$$
\begin{aligned}
r = \langle l^0, Tran^0, Gen^0, R^0, S^0\rangle &\xrightarrow{\mathcal{I}} \cdots \xrightarrow{\mathcal{I}} \langle l, Tran, Gen, R, S\rangle \\
&\xrightarrow{\mathcal{I}} \langle l', Tran', Gen', R, S\rangle
\end{aligned}
$$

where

$$
\wedge \begin{cases} \langle l^0, Tran^0, Gen^0, R^0, S^0\rangle \xrightarrow{\mathcal{I}} \cdots \xrightarrow{\mathcal{I}} \langle l, Tran, Gen, R, S\rangle \in RP \\ stmt_l \text{ is intraprocedural} \end{cases}
$$

Because

$$
\begin{aligned}
&\text{Let } \sigma = \langle l, Tran, Gen\rangle \text{ in} \\
&H^{\mathsf{e}} = dom(Tran') = dom(tran \circ Tran) = dom(Tran) \sqsubseteq \Pi(R)
\end{aligned}
$$

condition 7.11 holds.

Because

$$
\begin{aligned}
&\forall o^{\mathsf{e}} \in H^{\mathsf{e}} : Tran(o^{\mathsf{e}}) \subseteq \lceil o^{\mathsf{e}} \rceil^{\Pi(R)} \qquad // \text{ induction hypothesis} \\
\Rightarrow &\text{Let } \sigma = \langle l, Tran, Gen\rangle \text{ in} \\
&\forall o^{\mathsf{e}} \in H^{\mathsf{e}} : \\
&Tran'(o^{\mathsf{e}}) = tran(Tran(o^{\mathsf{e}})) \subseteq \lceil Tran(o^{\mathsf{e}}) \rceil^{\Pi(R)} = \lceil o^{\mathsf{e}} \rceil^{\Pi(R)} \\
&\hfill // \text{ Proposition 7.3.3}
\end{aligned}
$$

condition 7.12 holds.

Because

$$
\begin{aligned}
&\text{Let } \sigma = \langle l, Tran, Gen\rangle \text{ in}\\
&\begin{cases}
\forall o \in Gen : \exists A \in \Pi(R) : o \subseteq A & \text{// induction hypothesis}\\
\quad \Rightarrow \forall o \in Gen : tran(o) \subseteq \lceil o \rceil^{\Pi(R)} & \text{// Proposition 7.3.3}\\
\forall o' \in gen(\sigma) : \exists A \in \Pi(R) : o' \subseteq A & \text{// Proposition 7.3.3}
\end{cases}\\
\Rightarrow &\forall o' \in Gen' = gen(\sigma) \cup map(tran)(Gen) : \exists A \in \Pi(R) : o' \subseteq A
\end{aligned}
$$

condition 7.13 holds.

(b) Next consider the case

$$
\begin{aligned}
r = \langle l^0, Tran^0, Gen^0, R^0, S^0\rangle &\xrightarrow{\mathcal{I}} \cdots \xrightarrow{\mathcal{I}} \langle l_q^{\mathsf{c}}, Tran_q^{\mathsf{c}}, Gen_q^{\mathsf{c}}, R^{\mathsf{u}}, S^{\mathsf{u}}\rangle\\
&\xrightarrow{\mathcal{I}} \langle l_p^{\mathsf{e}}, Tran_p^{\mathsf{e}}, Gen_p^{\mathsf{e}}, R^{\mathsf{d}}, S^{\mathsf{d}}\rangle
\end{aligned}
$$

where

$$
\wedge \begin{cases}
\langle l^0, Tran^0, Gen^0, R^0, S^0\rangle \xrightarrow{\mathcal{I}} \cdots \xrightarrow{\mathcal{I}} \langle l_q^{\mathsf{c}}, Tran_q^{\mathsf{c}}, Gen_q^{\mathsf{c}}, R^{\mathsf{u}}, S^{\mathsf{u}}\rangle \in RP\\
l_q^{\mathsf{c}}\text{: } x{=}p(y_0, \ldots, y_k)
\end{cases}
$$

Because

$$
\begin{aligned}
&H^{\mathsf{c}} = image(Tran^{\mathsf{c}}) \cup Gen^{\mathsf{c}} \sqsubseteq \Pi(R^{\mathsf{u}}) & \text{// induction hypothesis}\\
\Rightarrow &H^{\mathsf{e}} = dom(Tran^{\mathsf{e}}) \sqsubseteq \Pi(R^{\mathsf{d}}) & \text{// Proposition 7.3.4}\\
\Rightarrow &\forall o^{\mathsf{e}} \in H^{\mathsf{e}} : Tran^{\mathsf{e}}(o^{\mathsf{e}}) = o^{\mathsf{e}} \subseteq \lceil o^{\mathsf{e}} \rceil^{\Pi(R^{\mathsf{d}})}
\end{aligned}
$$

conditions 7.11 and 7.12 hold.

Condition 7.13 holds because $Gen^{\mathsf{e}} = \emptyset$.

(c) Next consider the case

$$
\begin{aligned}
r = \langle l^0, Tran^0, Gen^0, R^0, S^0\rangle &\xrightarrow{\mathcal{I}} \cdots \xrightarrow{\mathcal{I}} \langle l_q^{\mathtt{c}}, Tran_q^{\mathtt{c}}, Gen_q^{\mathtt{c}}, R^{\mathtt{u}}, S^{\mathtt{u}}\rangle \\
&\xrightarrow{\mathcal{I}} \langle l_p^{\mathtt{e}}, Tran_p^{\mathtt{e}}, Gen_p^{\mathtt{e}}, R^{\mathtt{d}}, S^{\mathtt{d}}\rangle \xrightarrow{\mathcal{I}} \cdots \xrightarrow{\mathcal{I}} \langle l_p^{\mathtt{x}}, Tran_p^{\mathtt{x}}, Gen_p^{\mathtt{x}}, R^{\mathtt{d}}, S^{\mathtt{d}}\rangle \\
&\xrightarrow{\mathcal{I}} \langle l_q^{\mathtt{r}}, Tran_q^{\mathtt{r}}, Gen_q^{\mathtt{r}}, R^{\mathtt{u}}, S^{\mathtt{u}}\rangle
\end{aligned}
$$

where

$$
\wedge \begin{cases}
\langle l^0, Tran^0, Gen^0, R^0, S^0\rangle \xrightarrow{\mathcal{I}} \cdots \xrightarrow{\mathcal{I}} \langle l_q^{\mathtt{c}}, Tran_q^{\mathtt{c}}, Gen_q^{\mathtt{c}}, R^{\mathtt{u}}, S^{\mathtt{u}}\rangle \in RP \\
l_q^{\mathtt{c}}\text{: } x{=}p(y_0, \ldots, y_k) \\
\langle l_p^{\mathtt{e}}, Tran_p^{\mathtt{e}}, Gen_p^{\mathtt{e}}, R^{\mathtt{d}}, S^{\mathtt{d}}\rangle \xrightarrow{\mathcal{I}} \cdots \xrightarrow{\mathcal{I}} \langle l_p^{\mathtt{x}}, Tran_p^{\mathtt{x}}, Gen_p^{\mathtt{x}}, R^{\mathtt{d}}, S^{\mathtt{d}}\rangle \in SLRP \\
l_p^{\mathtt{x}}\text{: exit}
\end{cases}
$$

Because

$$
\begin{aligned}
& H_q^{\mathtt{c}} = image(Tran_q^{\mathtt{c}}) \cup Gen_q^{\mathtt{c}} \sqsubseteq \Pi(R^{\mathtt{u}}) \qquad \text{// induction hypothesis} \\
\Rightarrow & H_q^{\mathtt{e}} = dom(Tran_q^{\mathtt{r}}) = dom(tran^{l_q^{\mathtt{c}}} \circ Tran_q^{\mathtt{c}}) = dom(Tran_q^{\mathtt{c}}) \sqsubseteq \Pi(R^{\mathtt{u}}) \\
& \qquad \text{// Corollary 7.3.7}
\end{aligned}
$$

condition 7.11 holds.

Because

$$
\begin{aligned}
& \forall o_q^{\mathtt{e}} \in H_q^{\mathtt{e}} : Tran_q^{\mathtt{c}}(o_q^{\mathtt{e}}) \subseteq \lceil o_q^{\mathtt{e}} \rceil^{\Pi(R^{\mathtt{u}})} \qquad \text{// induction hypothesis} \\
\Rightarrow & \forall o_q^{\mathtt{e}} \in H_q^{\mathtt{e}} : \\
& Tran_q^{\mathtt{r}}(o_q^{\mathtt{e}}) = tran^{l_q^{\mathtt{c}}}(Tran_q^{\mathtt{c}}(o_q^{\mathtt{e}})) \subseteq \lceil Tran_q^{\mathtt{c}}(o_q^{\mathtt{e}}) \rceil^{\Pi(R^{\mathtt{u}})} = \lceil o_q^{\mathtt{e}} \rceil^{\Pi(R^{\mathtt{u}})} \\
& \qquad \text{// Corollary 7.3.7}
\end{aligned}
$$

condition 7.12 holds.

Because

$$\begin{cases} \forall o_q^{\mathtt{c}} \in Gen_q^{\mathtt{c}} : \exists A^{\mathtt{u}} \in \Pi(R^{\mathtt{u}}) : o \subseteq A & \text{// induction hypothesis} \\ \quad \Rightarrow \forall o_q^{\mathtt{c}} \in Gen_q^{\mathtt{c}} : tran^{l_q^{\mathtt{c}}}(o_q^{\mathtt{c}}) \subseteq \lceil o_q^{\mathtt{c}} \rceil^{\Pi(R^{\mathtt{u}})} & \text{// Corollary 7.3.7} \\ \forall o_q^{\mathtt{r}} \in gen^{l_q^{\mathtt{c}}} : \exists A^{\mathtt{u}} \in \Pi(R^{\mathtt{u}}) : o_q^{\mathtt{r}} \subseteq A^{\mathtt{u}} & \text{// Corollary 7.3.7} \end{cases}$$

$$\Rightarrow \forall o_q^{\mathtt{r}} \in Gen_q^{\mathtt{r}} = gen^{l_q^{\mathtt{c}}} \cup map(tran^{l_q^{\mathtt{c}}})(Gen_q^{\mathtt{c}}) : \exists A^{\mathtt{u}} \in \Pi(R^{\mathtt{u}}) : o_q^{\mathtt{r}} \subseteq A^{\mathtt{u}}$$

condition 7.13 holds.

■

Corollary 7.3.10 (Soundness of Clipper as a Slicing Analysis) *Given alias relations $\overline{R}$ derived from $[\![prog]\!]^{\mathbb{R}}$ of program prog and any RP*

$$r = \langle l^0, Tran^0, Gen^0, R^0, S^0 \rangle \xrightarrow{\mathcal{I}} \cdots \xrightarrow{\mathcal{I}} \langle l, Tran, Gen, R, S \rangle$$

from an initial state of the entry method `main`*, the following condition*

$$\wedge \begin{cases} l^0 \text{ is the entry label of main} & \\ H^0 = dom(Tran^0) = \emptyset & (a) \\ \forall o^0 \in H^0 : Tran^0(o^0) = o^0 & (b) \\ Gen^0 = \emptyset & (c) \\ R^0 = R_{main} & (d) \end{cases}$$

implies

$$H^{e} = dom(Tran) \sqsubseteq \Pi(R) \tag{7.14}$$

$$\forall o^{e} \in H^{e} : Tran(o^{e}) \subseteq \lceil o^{e} \rceil^{\Pi(R)} \tag{7.15}$$

$$Gen \sqsubseteq \Pi(R) \tag{7.16}$$

Proof Because

$$\wedge \begin{cases} H^0 = dom(Tran^0) \sqsubseteq \Pi(R^0) & \text{// condition (a)} \\ \forall o^0 \in H^0 : Tran^0(o^0) = o^0 & \text{// condition (b)} \\ Gen^0 = \emptyset & \text{// condition (c)} \\ R_{main} \subseteq R^0 & \text{// condition (d)} \end{cases}$$

conditions 7.14, 7.15, and 7.16 hold according to Proposition 7.3.9. ■

8. ASYNCHRONOUS CONTROL FLOW ANALYSIS

As illustrated in Section 1.4, in asynchronous messaging, control flows are scattered into many cooperatively-triggered message handling functions, obscuring causal relation among them.

For example, Fig. 8.1 shows part of an (contrived) app leveraging the asynchronous messaging framework of Android (Fig. 1.5). In this framework, a message can be denoted by the values of its fields `target` and `what`. For example, $\{target = Foo, what = 1\}$ is such a message in Fig. 8.1 where a handler object of type `Foo` is denoted by its type `Foo`.

When handling message $\{target = Foo, what = 1\}$ at line 26, another message $\{target = Bar, what = 1\}$ is enqueued at line 33. When further handling message $\{target = Bar, what = 1\}$ at line 45, another message $\{target = Foo, what = 2\}$ is enqueued at line 56. Similarly, when handling message $\{target = Foo, what = 2\}$ and $\{target = Bar, what = 2\}$ (at lines 27 and 46, respectively), two more messages $\{target = Bar, what = 2\}$ and $\{target = Foo, what = 1\}$ are further enqueued (at lines 37 and 52, respectively).

The message enqueuing operations imply the causal relation among the corresponding message handling operations, as shown in Fig. 8.2 where the nodes denotes message handling operations of the corresponding messages and the edges denotes message enqueuing operations representing the causal relation between the source and target messages.

In this chapter, a modular shape analysis is designed and specified (Section 8.1) to build an asynchronous control flow graph capturing such implicit causal realtion among these message handling functions (Section 8.2).

```
class Foo extends Handler {
  static final Foo INSTANCE = new Foo();
  void handle(Message m) {
    int w = m.what;
    switch (w) {
    case 1: handleFoo1(); break;
    case 2: handleFoo2(); break;
    ...
    }
  }
  void handleFoo1() {
    Handler h = Bar.INSTANCE;
    h.send(1);
  }
  void handleFoo2() {
    Handler h = Bar.INSTANCE;
    h.send(2);
  }
}
class Bar extends Handler {
  static final Bar INSTANCE = new Bar();
  void handle(Message m) {
    int w = m.what;
    switch (w) {
    case 1: handleBar1(); break;
    case 2: handleBar2(); break;
    ...
    }
  }
  void handleBar1() {
    Handler h = Foo.INSTANCE;
    h.send(2);
  }
  void handleBar2() {
    Handler h = Foo.INSTANCE;
    h.send(1);
  }
}
```

Fig. 8.1.: Example code for asynchronous messaging.

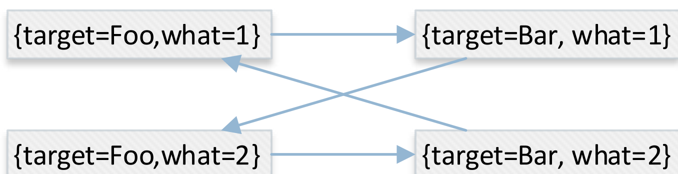


Fig. 8.2.: Asynchronous control flow graph of the example in Fig. 8.1. The nodes denotes message handling operations of the corresponding messages and the edges denotes message enqueuing operations representing the causal relation between the source and target messages.

8.1 ModShape

In this section, a modular shape analysis called ModShape is specified as an equation system. The fixed point of the equation system is a pair consisting of a heap invariance denoting the set of all possible message generated by the program and a trace invariance denoting the set of all possible execution traces of the program [39,40].

To specify ModShape, two new statements – pack and unpack – are introduced below:

$$\mathit{stmt} ::= \ldots \mid l\text{: pack}(x) \mid l\text{: } x = \text{unpack}()$$

In the asynchronous messaging example, the pack statement abstracts the message enqueuing operation, i.e. the invocation of `MessageQueue.enqueue()`, to extract the shape of the message object being enqueued. Dually, the unpack statement abstracts the message dequeuing operation, i.e. the invocation of `MessageQueue.next()`, to merge the shape of certain enqueued message into the current state σ. Fig. 8.3 shows the source code rewritten from that of Android's messaging framework with the invocation `mQueue.enqueue(m)` replaced by `pack(m)` (at line 7) and the invocation `m=mQueue.next()` replaced by `m=unpack()` (at line 15).

A message shape is defined as a packed heap. A packed heap consists of packed objects only and all access paths within the representation of a packed object are rooted at a pseudo header variable *hdr* [39] (as defined in Fig. 8.4).

Example 20 *The packed heap corresponding to the message* $\{target = Bar, what = 1\}$ *is shown in Fig. 8.5.*

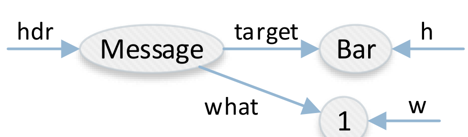
A packed heap is extraced from certain state σ via the pack operation $Pack(\cdot,\cdot,\cdot)$ defined in Fig. 8.6. The pack operation $Pack(\cdot,\cdot,\cdot)$ takes three parameters: an unpacked heap to extract the packed heap from, a local variable x, and a shape rim R representing a set of field paths such that all rim objects (i.e. objects $[x.\delta]_H$ for certain $\delta \in R$) and all intermediate objects (i.e. objects $[x.\delta_1]_H$ for certain $\delta_1.\delta_2 \in R$) are included in the packed heap.

```
abstract class Handler {
  MessageQueue mQueue;
  void send(int w) {
    Message m = new Message();
    m.target = this;
    m.what = w;
    pack(m); // mQueue.enqueue(m);
  }
  abstract void handle(Message m);
}
public class Looper {
  MessageQueue mQueue;
  void loop() {
    for (;;) {
      Message m = unpack(); // mQueue.next();
      Handler h = m.target;
      h.handle(m);
    }
  }
}
```

Fig. 8.3.: Rewritten code of Android's messaging framework.

$$
\begin{aligned}
&\text{pseudo header variable } hdr \in Var \\
&\text{packed object } \tilde{o} \in \widetilde{Obj} = 2^{hdr.\Delta} \setminus \{\emptyset\} \\
&\text{packed heap } \widetilde{H} \in \widetilde{Heap} \subset 2^{\widetilde{Obj}} \\
&\quad \text{s.t. } \forall \tilde{o}_1, \tilde{o}_2 \in \widetilde{H} : \tilde{o}_1 \cap \tilde{o}_2 \neq \emptyset \Rightarrow \tilde{o}_1 = \tilde{o}_2
\end{aligned}
$$

Fig. 8.4.: Header variable, packed object, and packed heap.



(a) Store-based heap.

$$
\begin{aligned}
Message &= \{hdr\} \\
Bar &= \{hdr.target\} \\
1 &= \{hdr.what\}
\end{aligned}
$$

(b) Storeless heap.

Fig. 8.5.: Packed heap of the message $\{target = Bar, what = 1\}$.

shape rim $R \in Rim = 2^{\Delta}$

pack operation $Pack(\cdot,\cdot,\cdot) : Heap \times Var \times Rim \rightarrow \widetilde{Heap}$

s.t. $\forall H \in Heap, x \in Var, R \in Rim$:

$Pack(H, x, R) \triangleq$

$$\text{let } \begin{cases} H^{pack} = \{[x.\delta_1]_H \mid \delta_1.\delta_2 \in R\} \\ pack = \lambda o \in H^{pack}.\{hdr.\delta \mid x.\delta \in o\} \end{cases} \quad \text{in}$$

$map(pack)(H^{pack})$

Fig. 8.6.: Shape rim and pack operation.

Example 21 *In the example program of Fig. 8.1, when handling the message* $\{target = Foo, what = 1\}$ *in method* ***handleFoo1()****, another message* $\{target = Bar, what = 1\}$ *is enqueued, as shown in Fig. 8.7. Applying the pack operation* $Pack(\cdot,\cdot,\cdot)$ *to the state at line 7 generates the packed heap in Fig. 8.5.*

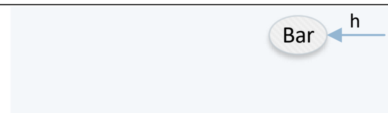
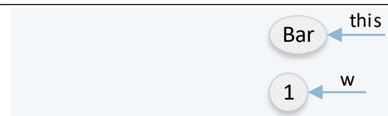
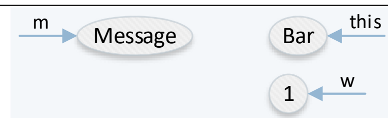
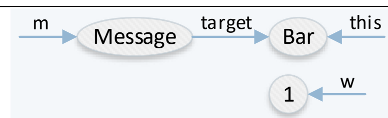
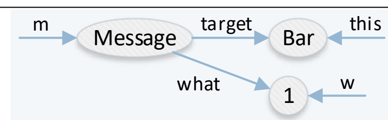
	Statement	Store-Based Heap	Storeless Heap
σ_1	32: h = Bar.INSTANCE		
σ_2	33: h.send(1)		$Bar = \{h\}$
σ_3	4: m = new Message()		$Bar = \{this\}$ $1 = \{w\}$
σ_4	5: m.target = this		$Message = \{m\}$ $Bar = \{this\}$ $1 = \{w\}$
σ_5	6: m.what = w		$Message = \{m\}$ $Bar = \{m.target, this\}$ $1 = \{w\}$
σ_6	7: pack(m)		$Message = \{m\}$ $Bar = \{m.target, this\}$ $1 = \{m.what, w\}$

Fig. 8.7.: Example illustrating packing.

Locally, the pack statement has no effect on the current execution state, as indicated by the definition of the pack transition $\cdot \xrightarrow{\mathcal{M}_{pack}} \cdot$ shown in Fig. 8.8. Different from [39], ownership transfer is not handled in ModShape. Instead, it is assumed that ownership transfer is implemented properly in the target program.

Given certain packed heap $\widetilde{H}$, an unpack statement $x = unpack()$ unpacks $\widetilde{H}$ by replacing the pseudo header variable *hdr* within the representation of the packed

$$\text{pack transition} \cdot \xrightarrow{\mathcal{M}_{pack}} \cdot \triangleq$$
$$\{\big\langle\langle l, Tran, Gen\rangle, \langle l{+}1, Tran, Gen\rangle\big\rangle \mid l\colon pack(x)\}$$

Fig. 8.8.: Pack transition of ModShape analysis.

objects from $\widetilde{H}$ with variable x on the left hand side of the unpack statement, as specified in Fig. 8.9.

$$
\begin{array}{l}
\text{unpack transition } \cdot \xrightarrow{\mathcal{M}_{unpack}(\cdot)} \cdot : \widetilde{Heap} \to \Sigma \times \Sigma \\
\text{s.t. } \forall \widetilde{H} \in \widetilde{Heap} : \cdot \xrightarrow{\mathcal{M}_{unpack}(\widetilde{H})} \cdot \triangleq \\
\left\{ \begin{array}{l}
\langle \langle l, Tran, Gen \rangle, \langle l{+}1, Tran, Gen' \rangle \rangle \mid \\
\quad l \colon x = \text{unpack}() \land \\
\quad \text{let } unpack = \lambda \tilde{o} \in \widetilde{H}.\{x.\delta \mid hdr.\delta \in \tilde{o}\} \text{ in} \\
\quad Gen' = Gen \cup map(unpack)(\widetilde{H})
\end{array} \right\}
\end{array}
$$

Fig. 8.9.: Unpack transition of ModShape analysis.

Example 22 *At line 15 of the example program in Fig. 8.1, the packed message in Fig. 8.5 is unpacked by replacing the pseudo header variable hdr within the packed message with variable m. For example, the representation of the* `Message` *object is transformed from* $\{hdr\}$ *to* $\{m\}$ *and the representation of the* `Bar` *object is transformed from* $\{hdr.target\}$ *to* $\{m.target\}$.

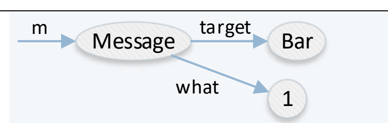
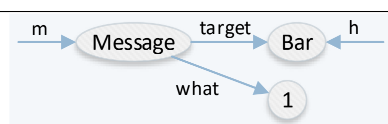
	Statement	Store-Based Heap	Storeless Heap
σ_1	15: m = unpack()		
σ_2	16: h = m.target		$Message = \{m\}$ $Bar = \{m.target\}$ $1 = \{m.what\}$
σ_3	17: h.handle(m)		$Message = \{m\}$ $Bar = \{m.target, h\}$ $1 = \{m.what\}$

Fig. 8.10.: Example illustrating unpacking.

For other intraprocedural statements, the transition relation $\cdot \xrightarrow{\mathcal{M}_{intra}} \cdot$ is the same as $\mathcal{LSL}$ semantics, as defined in Fig. 8.11.

$$\text{intraprocedural transition } \cdot \xrightarrow{\mathcal{M}_{intra}} \cdot \triangleq$$
$$\{\big\langle\langle l, Tran, Gen\rangle, \sigma'\big\rangle \mid stmt_l \text{ is intraprocedural } \wedge \langle l, Tran, Gen\rangle \rightarrow \sigma'\}$$

Fig. 8.11.: Other intraprocedural transition of ModShape analysis.

Similar to the $\mathcal{LSL}$ semantics, the interprocedural transition relation $\cdot \xrightarrow{\mathcal{M}_{inter}(\widetilde{tr}^{\mathtt{x}})} \cdot$ corresponding to certain exit trace $\widetilde{tr}^{\mathtt{x}}$ of the callee is defined as an instantiation of $\widetilde{tr}^{\mathtt{x}}$ at all possible call sites, as shown in Fig. 8.12.

$$
\begin{array}{l}
\text{ModShape trace } \widetilde{tr}, \sigma_0 \xrightarrow{\mathcal{M}} \sigma_1 \xrightarrow{\mathcal{M}} \cdots \xrightarrow{\mathcal{M}} \sigma_k \in \widetilde{Trace} \\
\quad \text{where } \sigma \xrightarrow{\mathcal{M}} \sigma' \text{ is one of } \begin{cases} \sigma \xrightarrow{\mathcal{M}_{intra}} \sigma' \\ \sigma \xrightarrow{\mathcal{M}_{inter}(\widetilde{tr}^{\mathtt{x}})} \sigma' \text{ for certain } \widetilde{tr}^{\mathtt{x}} \in \widetilde{Trace}^{\mathtt{x}} \\ \sigma \xrightarrow{\mathcal{M}_{pack}} \sigma' \\ \sigma \xrightarrow{\mathcal{M}_{unpack}(\widetilde{H})} \sigma' \text{ for certain } \widetilde{H} \in \widetilde{Heap} \end{cases} \\
\text{ModShape empty trace } \sigma \in \widetilde{Trace}^{0} \subset \widetilde{Trace} \\
\text{ModShape exit trace } \widetilde{Trace}^{\mathtt{x}} \triangleq \\
\quad \{\sigma^{\mathtt{e}} \xrightarrow{\mathcal{M}} \cdots \xrightarrow{\mathcal{M}} \langle l^{\mathtt{x}}, Tran^{\mathtt{x}}, Gen^{\mathtt{x}} \rangle \in \widetilde{Trace} \mid l^{\mathtt{x}}\text{: exit}\} \\
\text{interprocedural transition } \cdot \xrightarrow{\mathcal{M}_{inter}(\cdot)} \cdot : \widetilde{Trace}^{\mathtt{x}} \to \Sigma \times \Sigma \\
\quad \text{s.t. } \forall \widetilde{tr}^{\mathtt{x}} \in \widetilde{Trace}^{\mathtt{x}} : \cdot \xrightarrow{\mathcal{M}_{inter}(\widetilde{tr}^{\mathtt{x}})} \cdot \triangleq \\
\quad \left\{ \begin{array}{l} \big\langle \langle l^{\mathtt{c}}, Tran^{\mathtt{c}}, Gen^{\mathtt{c}} \rangle, \langle l^{\mathtt{r}}, Tran^{\mathtt{r}}, Gen^{\mathtt{r}} \rangle \big\rangle \mid \\ \quad l^{\mathtt{c}}\text{: } x{=}p(y_0, \ldots, y_k) \wedge \\ \quad \text{the declaration of } p \text{ is ``} t\ p(t_0\ h_0, \ldots, t_k\ h_k)\{body_p\}\text{''} \wedge \\ \quad \spadesuit \wedge \\ \quad \widetilde{tr}^{\mathtt{x}} = \langle l^{\mathtt{e}}, Tran^{\mathtt{e}}, Gen^{\mathtt{e}} \rangle \to \cdots \to \langle l^{\mathtt{x}}, Tran^{\mathtt{x}}, Gen^{\mathtt{x}} \rangle \wedge \\ \quad \clubsuit \end{array} \right\}
\end{array}
$$

Fig. 8.12.: Interprocedural transition of ModShape analysis.

The ModShape analysis computes two invariances: a trace invariance TI which is a set of ModShape traces and a packed heap invariance HI which is a set of packed heaps, as defined in Fig. 8.13.

Next, a top-down mapping function $\cdot \Downarrow$ is defined in Fig. 8.14 to help specifying the equation system of the ModShape analysis. Given a state $\langle l^{\mathtt{c}}, Tran^{\mathtt{c}}, Gen^{\mathtt{c}} \rangle$ at the call site $l^{\mathtt{c}}$: $x{=}p(y_0, \ldots, y_k)$, $\langle l^{\mathtt{c}}, Tran^{\mathtt{c}}, Gen^{\mathtt{c}} \rangle \Downarrow$ returns the state at the entry of the callee according the interprocedural $\mathcal{LSL}$ semantics defined in Fig. 7.15.

$$\text{trace invariance } TI \in TraceInv = 2^{\widetilde{Trace}}$$
$$\text{packed heap invariance } HI \in HeapInv = 2^{\widetilde{Heap}}$$

Fig. 8.13.: Trace invariance and packed heap invariance.

$$
\begin{aligned}
&\text{call state } \Sigma^{\mathsf{c}} \triangleq \{\langle l^{\mathsf{c}}, Tran^{\mathsf{c}}, Gen^{\mathsf{c}}\rangle \in \Sigma \mid l^{\mathsf{c}}\text{: } x{=}p(y_0,\ldots,y_k)\}\\
&\text{top-down mapping } \cdot\Downarrow\text{: } \Sigma^{\mathsf{c}} \to \Sigma\\
&\quad \text{s.t. } \forall \langle l^{\mathsf{c}}, Tran^{\mathsf{c}}, Gen^{\mathsf{c}}\rangle \in \Sigma^{\mathsf{c}} :\\
&\quad \langle l^{\mathsf{c}}, Tran^{\mathsf{c}}, Gen^{\mathsf{c}}\rangle\Downarrow\triangleq\\
&\quad \text{let} \begin{cases} l^{\mathsf{c}}\text{: } x{=}p(y_0,\ldots,y_k)\\ \text{the declaration of } p \text{ is ``} t\ p(t_0\ h_0,\ldots,t_k\ h_k)\{body_p\}\text{''} \\ \spadesuit \end{cases} \quad \text{in}\\
&\quad \langle l^{\mathsf{e}}, Tran^{\mathsf{e}}, Gen^{\mathsf{e}}\rangle
\end{aligned}
$$

Fig. 8.14.: Domains and helper functions of the ModShape analysis.

The ModShape equation system corresponding to a givan program *prog* and shape rim R is defined with a function F^R_{prog} which takes a trace invariance TI and packed heap invariance HI and return a new trace invariance TI' and packed heap invariance HI', as defined in Fig. 8.15.

Given certain program *prog* and shape rim R, let $F^R_{prog}(TI, HI) = \langle TI', HI' \rangle$ where

$$
TI' = \cup \begin{cases}
\{\sigma^0\} & (a) \\
\left\{ \begin{array}{l} \sigma^{\mathsf{e}} \xrightarrow{\mathcal{M}} \cdots \xrightarrow{\mathcal{M}} \sigma \xrightarrow{\mathcal{M}_{intra}} \sigma' \mid \\ \quad \sigma^{\mathsf{e}} \xrightarrow{\mathcal{M}} \cdots \xrightarrow{\mathcal{M}} \sigma \in TI \;\wedge\; \sigma \xrightarrow{\mathcal{M}_{intra}} \sigma' \end{array} \right\} & (b) \\
\left\{ \begin{array}{l} \sigma^{\mathsf{e}} \xrightarrow{\mathcal{M}} \cdots \xrightarrow{\mathcal{M}} \sigma^{\mathsf{c}} \xrightarrow{\mathcal{M}_{inter}(\widetilde{tr}^{\mathsf{x}})} \sigma^{\mathsf{r}} \mid \\ \quad \widetilde{tr}^{\mathsf{x}} = \sigma^{\mathsf{e}} \xrightarrow{\mathcal{M}} \cdots \xrightarrow{\mathcal{M}} \langle l^{\mathsf{x}}, Tran^{\mathsf{x}}, Gen^{\mathsf{x}} \rangle \in TI \;\wedge \\ \quad l^{\mathsf{x}}\text{: exit} \;\wedge \\ \quad \sigma^{\mathsf{c}} \xrightarrow{\mathcal{M}_{inter}(\widetilde{tr}^{\mathsf{x}})} \sigma^{\mathsf{r}} \end{array} \right\} & (c) \\
\left\{ \begin{array}{l} \sigma^{\mathsf{e}} \xrightarrow{\mathcal{M}} \cdots \xrightarrow{\mathcal{M}} \sigma \xrightarrow{\mathcal{M}_{pack}} \sigma' \mid \\ \quad \sigma^{\mathsf{e}} \xrightarrow{\mathcal{M}} \cdots \xrightarrow{\mathcal{M}} \sigma \in TI \;\wedge\; \sigma \xrightarrow{\mathcal{M}_{pack}} \sigma' \end{array} \right\} & (d) \\
\left\{ \sigma^{\mathsf{e}} \xrightarrow{\mathcal{M}} \cdots \xrightarrow{\mathcal{M}} \sigma \xrightarrow{\mathcal{M}_{unpack}(\widetilde{H})} \sigma' \;\middle|\; \begin{array}{l} \sigma^{\mathsf{e}} \xrightarrow{\mathcal{M}} \cdots \xrightarrow{\mathcal{M}} \sigma \in TI \;\wedge \\ \widetilde{H} \in HI \;\wedge\; \sigma \xrightarrow{\mathcal{M}_{unpack}(\widetilde{H})} \sigma' \end{array} \right\} & (e) \\
\left\{ \langle l^{\mathsf{c}}_q, Tran^{\mathsf{c}}_q, Gen^{\mathsf{c}}_q \rangle \Downarrow \;\middle|\; \begin{array}{l} \sigma^{\mathsf{e}}_q \xrightarrow{\mathcal{M}} \cdots \xrightarrow{\mathcal{M}} \langle l^{\mathsf{c}}_q, Tran^{\mathsf{c}}_q, Gen^{\mathsf{c}}_q \rangle \in TI \;\wedge \\ l^{\mathsf{c}}_q\text{: } x{=}p(y_0, \ldots, y_k) \end{array} \right\} & (f)
\end{cases}
$$

$$
HI' = \left\{ \widetilde{H} \;\middle|\; \begin{array}{l} \sigma^{\mathsf{e}} \xrightarrow{\mathcal{M}} \cdots \xrightarrow{\mathcal{M}} \langle l, Tran, Gen \rangle \in TI \;\wedge \\ l\text{: pack}(x) \;\wedge \\ \triangle \begin{cases} H = image(Tran) \cup Gen \;\wedge \\ \widetilde{H} = Pack(H, x, R) \end{cases} \end{array} \right\}
$$

Fig. 8.15.: Equation system of the ModShape analysis.

The new trace invariance TI' consists of the six kinds of traces:

1. The empty trace consisting of the initial execution state σ^0 (case (a) of F^R_{prog});

2. Traces obtained by extending existing traces in TI with intraprocedural transitions $\cdot \xrightarrow{\mathcal{M}_{intra}} \cdot$ (case (b) of F^R_{prog});

3. Traces obtained by extending existing traces in TI with interprocedural transitions $\cdot \xrightarrow{\mathcal{M}_{inter}(\widetilde{tr}^{\mathtt{x}})} \cdot$ corresponding to certain exit trace $\widetilde{tr}^{\mathtt{x}}$ in TI (case (c) of F^R_{prog});

4. Traces obtained by extending existing traces in TI with transitions $\cdot \xrightarrow{\mathcal{M}_{pack}} \cdot$ corresponding to pack statements (case (d) of F^R_{prog});

5. Traces obtained by extending existing traces in TI with transitions $\cdot \xrightarrow{\mathcal{M}_{unpack}(\widetilde{H})} \cdot$ unpacking certain packed heap $\widetilde{H}$ in HI (case (e) of F^R_{prog});

6. The empty trace consisting of the callee entry state $\langle l^{\mathtt{c}}_q, Tran^{\mathtt{c}}_q, Gen^{\mathtt{c}}_q \rangle \Downarrow$ with respect to the caller state $\langle l^{\mathtt{c}}_q, Tran^{\mathtt{c}}_q, Gen^{\mathtt{c}}_q \rangle$ at $l^{\mathtt{c}}_q$: $x{=}p(y_0, \ldots, y_k)$ reachable via certain trace in TI (case (f) of F^R_{prog}).

The new packed heap invariance HI' is obtained by extracting packed heap from all states $\langle l, Tran, Gen \rangle$ at "l: $pack(x)$" reachable via certain trace in TI.

The least fixed point of F^R_{prog} is denoted by $\mathtt{lfp}(F^R_{prog}) = \langle TI^R_{prog}, HI^R_{prog} \rangle$.

8.2 Asynchronous Control Flow Analysis

In asynchronous messaging, handling certain messages could cause more messages being handled. For example, Fig. 8.16 shows execution traces of the program in Fig. 8.1. In these traces, the handling of message $\{target = Bar, what = 1\}$ leads to the enqueuing of message $\{target = Foo, what = 2\}$, causing the latter to be handled asynchronously. Such causal relation between the handling of different messages can be modeled as a directed graph called asynchronous control flow graph ($ACFG$). The nodes within an $ACFG$ corresponds to packed heaps encoding messages and the edges between nodes denote that the handling of the source message enqueues (and thus causes the handling of) the target message, as shown in the example $ACFG$ in Fig. 8.2.

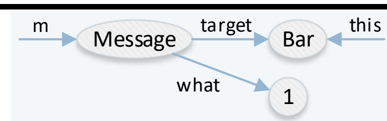
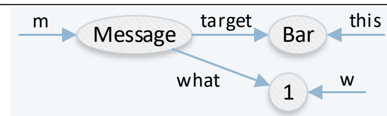
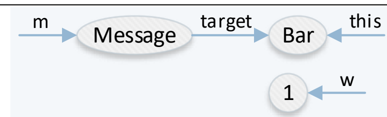
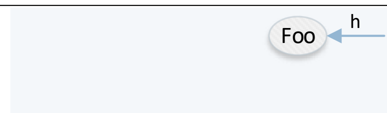
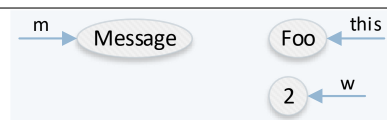
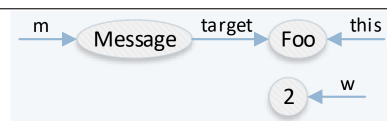
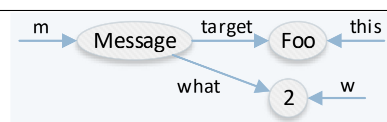
	Statement	Store-Based Heap	Storeless Heap
σ_4	43: w = m.what		$Message = \{m\}$ $Bar = \{m.target, this\}$ $1 = \{m.what\}$
σ_5	44: switch(w)		$Message = \{m\}$ $Bar = \{m.target, this\}$ $1 = \{m.what, w\}$
σ_6	45: handleBar1()		$Message = \{m\}$ $Bar = \{m.target, this\}$ $1 = \{m.what, w\}$
σ_7	51: h = Foo.INSTANCE		
σ_8	52: h.send(2)		$Foo = \{h\}$
σ_9	4: m = new Message()		$Foo = \{this\}$ $2 = \{w\}$
σ_{10}	5: m.target = this		$Message = \{m\}$ $Foo = \{this\}$ $2 = \{w\}$
σ_{11}	6: m.what = w		$Message = \{m\}$ $Foo = \{m.target, this\}$ $2 = \{w\}$
σ_{12}	7: pack(m)		$Message = \{m\}$ $Foo = \{m.target, this\}$ $2 = \{m.what, w\}$

Fig. 8.16.: Example illustrating asynchronous control flow analysis. The thick lines separate different traces.

An $ACFG$ is encoded as a set of such edges between source/target messages, i.e.

$$ACFG \in 2^{\widetilde{Heap} \times \widetilde{Heap}}$$

Example 23 *The asynchronous call graph of Fig. 8.2 generated by the program in Fig. 8.1 is denoted by*

$$ACFG = \left\{ \begin{array}{l} \langle \{target = Foo, what = 1\}, \{target = Bar, what = 1\} \rangle, \\ \langle \{target = Bar, what = 1\}, \{target = Foo, what = 2\} \rangle, \\ \langle \{target = Foo, what = 2\}, \{target = Bar, what = 2\} \rangle, \\ \langle \{target = Bar, what = 2\}, \{target = Foo, what = 1\} \rangle \end{array} \right\}$$

.

Given a program *prog* and shape rim R outlining the shape of the messages, an $ACFG$ can be extracted from the trace invariance computed by the ModShape analysis. To achieve that, an intermediate result called enqueuing summary (the Enq defined below) is computed first via the equation system in Fig. 8.17.

$$Enq \in 2^{\{\langle l, Tran, Gen \rangle \mid l\text{: pack}(x) \vee l\text{: } x = p(y_0, \ldots, y_k)\} \times \widetilde{Heap}}$$

An enqueuing summary is a set of $\langle \text{execution state}, \text{packed heap} \rangle$ pairs where a pair of the form $\langle \langle l, Tran, Gen \rangle, \widetilde{H} \rangle$ indicates that the statement $stmt_l$ extracts and enqueues a message denoted by the packed heap $\widetilde{H}$ from the current execution state $\langle l, Tran, Gen \rangle$. The statement $stmt_l$ could be a pack statement "l: pack(x)" that extracts and enqueues the message *directly* (case (a) of G^R_{prog}) or a call statement "l: x=$p(y_0, \ldots, y_k)$" that extracts and enqueues the message *indirectly* via the pack statements within the (transitively) invoked callees (case (b) of G^R_{prog}).

The least fixed point of G^R_{prog} is denoted by $\texttt{lfp}(G^R_{prog}) = Enq^R_{prog}$.

Example 24 *In the execution traces of Fig. 8.16, the pack statement at line 7 directly extracts and enqueues the message* $\{target = Foo, what = 2\}$ *from current state* σ_{12}.

Given certain program *prog* and message shape rim R, let $G^{R}_{prog}(Enq) = Enq'$ where

$$
\begin{array}{l}
Enq' = \\
\quad \left\{ \begin{array}{l}
\left\{ \langle\langle l, Tran, Gen\rangle, \widetilde{H}\rangle \;\middle|\; \begin{array}{l} \sigma^{\mathsf{e}} \xrightarrow{\mathcal{M}} \cdots \xrightarrow{\mathcal{M}} \langle l, Tran, Gen\rangle \in TI^{R}_{prog} \wedge \\ l\text{: pack}(x) \wedge \\ \triangle \end{array} \right\} \qquad (a) \\
\cup \left\{ \begin{array}{l}
\langle \sigma^{\mathsf{c}}_{q}, \widetilde{H}\rangle \mid \\
\quad \sigma^{\mathsf{e}}_{p} \xrightarrow{\mathcal{M}} \cdots \xrightarrow{\mathcal{M}} \sigma^{1}_{p} \xrightarrow{\mathcal{M}} \sigma^{2}_{p} \in TI^{R}_{prog} \wedge \\
\quad \langle \sigma^{2}_{p}, \widetilde{H}\rangle \in Enq \wedge \\
\quad \nexists \langle l_p, Tran_p, Gen_p\rangle \in \sigma^{\mathsf{e}}_{p} \xrightarrow{\mathcal{M}} \cdots \xrightarrow{\mathcal{M}} \sigma^{1}_{p} : l_p\text{: } x = \text{unpack}() \wedge \\
\quad \sigma^{\mathsf{e}}_{q} \xrightarrow{\mathcal{M}} \cdots \xrightarrow{\mathcal{M}} \langle l^{\mathsf{c}}_{q}, Tran^{\mathsf{c}}_{q}, Gen^{\mathsf{c}}_{q}\rangle \in TI^{R}_{prog} \wedge \\
\quad l^{\mathsf{c}}_{q}\text{: } x{=}p(y_0, \ldots, y_k) \wedge \\
\quad \sigma^{\mathsf{e}}_{p} = \langle l^{\mathsf{c}}_{q}, Tran^{\mathsf{c}}_{q}, Gen^{\mathsf{c}}_{q}\rangle \Downarrow
\end{array} \right\} \qquad (b)
\end{array} \right.
\end{array}
$$

Fig. 8.17.: Equation system of enqueuing summary.

Hence $\langle \sigma_{12}, \{target = Foo, what = 2\}\rangle \in Enq^R_{prog}$*. It follows that the method call (at line 45) of current method indirectly extracts and enqueues the same message from the state* σ_6*. Hence* $\langle \sigma_6, \{target = Foo, what = 2\}\rangle \in Enq^R_{prog}$*. Transitively the method call (at line 17 of Fig. 8.3) also indirectly extracts and enqueues the same message from the state* σ_3 *of Fig. 8.10. Hence* $\langle \sigma_3, \{target = Foo, what = 2\}\rangle \in Enq^R_{prog}$*.*

Given certain program *prog* and message shape rim R, the asynchronous control flow graph $ACFG^R_{prog}$ can be extracted from the trace invariance TI^R_{prog} and enqueuing summary Enq^R_{prog}, i.e. an asynchronous control flow edge $\langle \widetilde{H}, \widetilde{H}'\rangle \in ACFG^R_{prog}$ if there is a trace $\widetilde{tr} \in TI^R_{prog}$ along which an unpack statement dequeuing and unpacking a packed message heap $\widetilde{H}$ leads to a pack statement extracting and enqueuing a packed message heap $\widetilde{H}'$, as specified in Fig. 8.18.

Given certain program *prog* and message shape rim R, let

$$ACFG^R_{prog} = \left\{ \langle \widetilde{H}, \widetilde{H}'\rangle \;\middle|\; \begin{array}{l} \sigma^{\mathsf{e}} \xrightarrow{\mathcal{M}} \cdots \xrightarrow{\mathcal{M}} \langle l, Tran, Gen\rangle \xrightarrow{\mathcal{M}} \sigma_1 \xrightarrow{\mathcal{M}} \cdots \xrightarrow{\mathcal{M}} \sigma_2 \in TI^R_{prog} \wedge \\ \langle \sigma_2, \widetilde{H}'\rangle \in Enq^R_{prog} \wedge \\ \nexists \langle l_3, Tran_3, Gen_3\rangle \in \sigma_1 \xrightarrow{\mathcal{M}} \cdots \xrightarrow{\mathcal{M}} \sigma_2 : l_3\text{: } x = \text{unpack}() \wedge \\ \triangle \end{array} \right\}$$

Fig. 8.18.: Asynchronous control flow analysis.

Example 25 *In the execution trace of Fig. 8.10, the unpack statement at line 15 dequeues and unpacks the message* $\{target = Bar, what = 1\}$ *and the following call statement at line 17 indirectly extracts and enqueues the message* $\{target = Foo, what = 2\}$*, as illustrated in Example 24. Hence the asynchronous control flow edge* $\langle \{target = Bar, what = 1\}, \{target = Foo, what = 2\}\rangle \in ACFG^R_{prog}$*, as shown in Fig. 8.2.*

An evaluation of ModShape for asynchronous control flow analysis is able to build an asynchronous control flow graph consisting of 52 nodes (message types) within two minutes from the Android framework of version 2.3.7_r1.

9. RELATED WORK

Sridharan et al. [5] also proposed a demand-driven approach to points-to analysis. Different from DynaSens where Clipper is used as a slicing analysis to refine context-sensitivity of the points-to analysis, in their approach context and heap are both modeled with context-free languages (*CFL*) and thus resorts to over-approximation when handling recursive method invocations. As explained in Section 3.0.2, this approach suffers from precision loss.

Rountev et al. [41] addressed the scalability challenge for interprocedural distributive environment (*IDE*) dataflow problems on large libraries with pre-computed library summary information. Although the authors claim that the proposed approach reduces significantly the cost of whole-program IDE analyses, their approach is essentially based on a context-insensitive heap model where objects are approximatelyax modeled with their types - leading to precision loss when tracking data flows through the heap. The demand-driven approach implemented with Clipper is based on a storeless heap model where objects are precisely modeled with alias classes of access paths capable of tracking data flows through the heap without loss of precision.

Cunningham et al. [11] proposed the Explicit Event Library called *libeel* which provides a unified interface for registering, canceling, and dispatching callbacks. This design simplified the task of implicit control flow analysis because callback registrations can only be carried out by invoking this interface where the registered callback methods are explicitly specified. In most real-world programs, callback registrations can be carried out by many different customized interfaces or as side effects of any interfaces provided by the framework. The implicit control flow analysis implemented with Clipper is more generally applicable to almost all existing frameworks.

10. SUMMARY

The traditional model of homogeneous whole-program analysis has several limitations that make it unsuitable for real-world programs built on large scale frameworks. Particularly, the imprecision in resolving heap-carried dependency hindered the application of precise but expensive analyses to these programs. The research impact of such analyses can be broadened significantly if this limitation is resolved. As a step towards achieving this goal, in this thesis we proposed a slicing method for resolving heap-carried dependency and three client analyses demonstrating how to employ such dependence information to build precise and scalable client analyses.

Our slicing method (Chapter 4) is access path and tabulation based. Access path based heap abstraction strikes a balance between scalability and precision, both are necessary to extract useful information from large scale programs to bootstrap expensive client analyses. Tabulation based approach is necessary to handle interprocedural data flow without precision loss, especially in the presence of recursive invocations.

One application of our slicing method, the demand-driven refinement of points-to analysis (Chapter 6), provides a long due solution to the dilemma of trade-off between precision and scalability in context-sensitive points-to analysis. By identifying a subset of the program elements relevant to the flow of interest, our slicing analysis can automatically improve the precision of the points-to analysis by keeping more context information on these elements, as well as the scalability of it by keeping less context information on others.

Another application of the slicing method to identify the callback method (or registration call site) with respect to certain registration call site (or callback method), presented in Chapter 5, provides a tool to help programmer understand the interaction between the framework and application plug-ins and to extract a concise but precise model of the framework to improve the scalability when analyzing the application plug-ins and the extracted framework model as a whole.

The third application of the slicing method enables certain flow sensitive shape analysis on large scale program frameworks to resolve causal relations among messages introduced by asynchronous message passing (Chapter 8). These causal relations capture control flows implicitly, which are necessary for works on data race detection, type-state verification, etc.

REFERENCES

REFERENCES

[1] P. Naur and B. Randell, Eds., *Software Engineering: Report of a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division, NATO*, 1969.

[2] A. Milanova, A. Rountev, and B. G. Ryder, "Parameterized object sensitivity for points-to analysis for java," *ACM Trans. Softw. Eng. Methodol.*, vol. 14, no. 1, pp. 1–41, Jan. 2005. [Online]. Available: http://doi.acm.org/10.1145/1044834.1044835

[3] T. W. Reps, "Solving demand versions of interprocedural analysis problems," in *Proceedings of the 5th International Conference on Compiler Construction*, ser. CC '94. London, UK, UK: Springer-Verlag, 1994, pp. 389–403. [Online]. Available: http://dl.acm.org/citation.cfm?id=647472.727424

[4] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, pp. 319–349, Jul. 1987. [Online]. Available: http://doi.acm.org/10.1145/24039.24041

[5] M. Sridharan and R. Bodík, "Refinement-based context-sensitive points-to analysis for java," in *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '06. New York, NY, USA: ACM, 2006, pp. 387–400. [Online]. Available: http://doi.acm.org/10.1145/1133981.1134027

[6] G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems.* Cambridge, MA, USA: MIT Press, 1986.

[7] J. Armstrong, "Erlang - a survey of the language and its industrial applications," in *In Proceedings of the symposium on industrial applications of Prolog (INAP96). 16-18*, 1996.

[8] R. Virding, C. Wikström, and M. Williams, *Concurrent Programming in ERLANG (2Nd Ed.)*, J. Armstrong, Ed. Hertfordshire, UK, UK: Prentice Hall International (UK) Ltd., 1996.

[9] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, *The Java Virtual Machine Specification, Java SE 8 Edition*, 1st ed. Addison-Wesley Professional, 2014.

[10] P. Haller and M. Odersky, "Scala actors: Unifying thread-based and event-based programming," *Theor. Comput. Sci.*, vol. 410, no. 2-3, pp. 202–220, Feb. 2009. [Online]. Available: http://dx.doi.org/10.1016/j.tcs.2008.09.019

[11] R. Cunningham and E. Kohler, "Making events less slippery with eel," in *Proceedings of the 10th Conference on Hot Topics in Operating Systems - Volume 10*, ser. HOTOS'05. Berkeley, CA, USA: USENIX Association, 2005, pp. 3–3. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251123.1251126

[12] N. Rinetzky, J. Bauer, T. Reps, M. Sagiv, and R. Wilhelm, "A semantics for procedure local heaps and its abstractions," in *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '05. New York, NY, USA: ACM, 2005, pp. 296–309. [Online]. Available: http://doi.acm.org/10.1145/1040305.1040330

[13] P. Liang and M. Naik, "Scaling abstraction refinement via pruning," in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: ACM, 2011, pp. 590–601. [Online]. Available: http://doi.acm.org/10.1145/1993498.1993567

[14] X. Zhang, R. Mangal, R. Grigore, M. Naik, and H. Yang, "On abstraction refinement for program analyses in datalog," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: ACM, 2014, pp. 239–248. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594327

[15] O. Shivers, "Control flow analysis in scheme," in *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, ser. PLDI '88. New York, NY, USA: ACM, 1988, pp. 164–174. [Online]. Available: http://doi.acm.org/10.1145/53990.54007

[16] M. Sharir and A. Pnueli, "Two approaches to interprocedural data flow analysis," 1978.

[17] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: ACM, 2014, pp. 259–269. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594299

[18] I. Matosevic and T. S. Abdelrahman, "Efficient bottom-up heap analysis for symbolic path-based data access summaries," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, ser. CGO '12. New York, NY, USA: ACM, 2012, pp. 252–263. [Online]. Available: http://doi.acm.org/10.1145/2259016.2259049

[19] J. M. Lucassen and D. K. Gifford, "Polymorphic effect systems," in *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '88. New York, NY, USA: ACM, 1988, pp. 47–57. [Online]. Available: http://doi.acm.org/10.1145/73560.73564

[20] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of Program Analysis.* Springer Publishing Company, Incorporated, 2010.

[21] H. R. Nielson and F. Nielson, *Semantics with Applications: A Formal Introduction.* New York, NY, USA: John Wiley & Sons, Inc., 1992.

[22] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '95. New York, NY, USA: ACM, 1995, pp. 49–61. [Online]. Available: http://doi.acm.org/10.1145/199448.199462

[23] R. Lafore, *Data Structures and Algorithms in Java*, 2nd ed. Indianapolis, IN, USA: Sams, 2002.

[24] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, & Tools with Gradiance*, 2nd ed. USA: Addison-Wesley Publishing Company, 2007.

[25] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*, 3rd ed. The MIT Press, 2009.

[26] Y. Cao, Y. Fratantonio, A. Bianchi, M. Egele, C. Krügel, G. Vigna, and Y. Chen, "Edgeminer: Automatically detecting implicit control flow transitions through the android framework," in *NDSS*, 2015.

[27] S. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay, "Effective typestate verification in the presence of aliasing," in *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, ser. ISSTA '06. New York, NY, USA: ACM, 2006, pp. 133–144. [Online]. Available: http://doi.acm.org/10.1145/1146238.1146254

[28] Y. Smaragdakis, G. Kastrinis, and G. Balatsouras, "Introspective analysis: Context-sensitivity, across the board," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: ACM, 2014, pp. 485–495. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594320

[29] Y. Smaragdakis, M. Bravenboer, and O. Lhoták, "Pick your contexts well: Understanding object-sensitivity," in *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '11. New York, NY, USA: ACM, 2011, pp. 17–30. [Online]. Available: http://doi.acm.org/10.1145/1926385.1926390

[30] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, and McKinley, "The DaCapo benchmarks: Java benchmarking development and analysis," in *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*. New York, NY, USA: ACM Press, Oct. 2006, pp. 169–190.

[31] E. Duesterwald, R. Gupta, and M. L. Soffa, "Demand-driven computation of interprocedural data flow," in *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '95. New York, NY, USA: ACM, 1995, pp. 37–48. [Online]. Available: http://doi.acm.org/10.1145/199448.199461

[32] N. Heintze and O. Tardieu, "Demand-driven pointer analysis," in *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, ser. PLDI '01. New York, NY, USA: ACM, 2001, pp. 24–34. [Online]. Available: http://doi.acm.org/10.1145/378795.378802

[33] S. Blackshear, A. Gendreau, and B.-Y. E. Chang, "Droidel: A general approach to android framework modeling," in *Proceedings of the 4th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, ser. SOAP 2015. New York, NY, USA: ACM, 2015, pp. 19–25. [Online]. Available: http://doi.acm.org/10.1145/2771284.2771288

[34] M. Sagiv, T. Reps, and R. Wilhelm, "Solving shape-analysis problems in languages with destructive updating," *ACM Trans. Program. Lang. Syst.*, vol. 20, no. 1, pp. 1–50, Jan. 1998. [Online]. Available: http://doi.acm.org/10.1145/271510.271517

[35] H. Jonkers, "Abstract storage structures. in de bakker and van vllet, editors, algorithmic languages," 1981.

[36] A. Deutsch, "A storeless model of aliasing and its abstractions using finite representations of right-regular equivalence relations," in *Proceedings of the 1992 International Conference on Computer Languages*, Apr 1992, pp. 2–13.

[37] R. E. Strom and S. Yemini, "Typestate: A programming language concept for enhancing software reliability," *IEEE Transactions on Software Engineering*, vol. SE-12, no. 1, pp. 157–171, Jan 1986.

[38] K. Hrbacek and T. Jech, *Introduction to set theory.* New York: Marcel Dekker, 1999.

[39] N. Rinetzky, A. Poetzsch-Heffter, G. Ramalingam, M. Sagiv, and E. Yahav, "Modular shape analysis for dynamically encapsulated programs," in *Programming Languages and Systems*, R. De Nicola, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 220–236.

[40] A. Gotsman, J. Berdine, B. Cook, and M. Sagiv, "Thread-modular shape analysis," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07. New York, NY, USA: ACM, 2007, pp. 266–277. [Online]. Available: http://doi.acm.org/10.1145/1250734.1250765

[41] A. Rountev, M. Sharp, and G. Xu, "Ide dataflow analysis in the presence of large object-oriented libraries," in *Compiler Construction*, L. Hendren, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 53–68.

VITA

VITA

Chenguang Sun was born in Dalian, China. He received a B.S. in Computer Science from Tsinghua University, Beijing, China.

From 2008 to 2018, he was a Research Assistant with School of Electrical and Computer Engineering at Purdue University. He joined Google Inc. in 2019.