Toward Interprocedural Pointer and Effect Analysis for Scala

by

Etienne Kneuss

BSc., Computer Science
École Polytechnique Fédérale de Lausanne (2009)

Submitted to the School of Computer and Communication Sciences
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

at the

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

August 2011

Author .....
School of Computer and Communication Sciences
June 24, 2011

Toward Interprocedural Pointer and Effect Analysis for Scala

by

Etienne Kneuss

Submitted to the School of Computer and Communication Sciences
on June 24, 2011, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science

Abstract

Static program analysis techniques working on object-oriented languages require precise knowledge of the aliasing relation between variables. This knowledge is important to, among other things, understand the read and write effects of method calls on objects. Understanding such effects in turn enables compiler optimizations and other code transformations such as automated parallelization. This thesis presents a combination of a pointer analysis with a memory effect analysis for the Scala programming language. Our analysis is based on abstract interpretation, and computes summaries of method effects as graphs. This representation allows the analysis to be compositional. Our second contribution is an implementation of our analysis in a tool called Insane. Our tool is built as a plugin for the official Scala compiler. It accepts any Scala program, and is freely available.

Thesis Supervisor: Viktor Kuncak
Title: Assistant Professor

Thesis Supervisor: Philippe Suter
Title: Ph.D. Student

Acknowledgments

First and foremost I would like to thank my supervisor, Viktor Kuncak, for his sustained enthousiasm, inspired suggestions and exemplary guidance thourough the course of this thesis. I would also like to thank him for giving me the opportunity to work on such interesting subjects, and look forward to working with him as I continue with my research. I would also like to thank Philippe Suter for his continuous feedback and guidance without which this thesis would not have been possible.

I would like to extend my sincere thanks to the members of the Laboratory for Automated Reasoning and Analysis for numerous stimulating discussions: Ali, Andrej, Eva, Giuliano, Hossein, Ruzica, Swen and Tihomir: thanks. I would also like to thank Lukas Rytz for his patience and technical support regarding the Scala compiler. Finally I thank those who continuously supported me; my family, my friends, and of course Aline.

Contents

Chapter 1

Introduction

Pointer analysis is a static analysis technique that builds information on the relations between pointers and allocated objects. It is also often referred to as points-to or alias analysis. In object-oriented languages such as Scala, the use of pointers is pervasive, rendering even basic static analyses techniques brittle. It is thus often necessary to establish information on the aliasing relations between variables, as well as some knowledge of the shape of structures stored on the heap. This then enables opportunities to run more analyses or to perform compiler optimizations.

Effect analyses attempt to summarize the side effects of procedures in a certain domain. In this thesis, we focus on memory-based effects, and are thus interested in computing a summary of read and write operations performed on object fields. Clearly, any such effect analysis needs to rely on a good pointer analysis, and vice-versa. For this reason, we perform both analyses side by side.

The summary of effects coupled with aliasing information can later be used to perform various kinds of optimizations or enable more sophisticated analyses. For instance, if we establish that two sequential operations affect disjoint parts of the heap, we could safely run them in parallel. Also, given a precise alias information, we could perform some form of typestate analysis, which consists in checking that objects are used following a certain protocol. A typical example is objects representing files: it is required that you first open a file before reading from it. Such analyses[FYD+08] require a precise alias analysis to limit the amount of spurious warnings.

Our analysis is based on abstract interpretation [CC77, CC02]. The abstract representation consists of graphs and is closely based on [Sal06]. Such graphs are built so that the analysis is compositional. One of the challenges of such analysis is to provide a representation for the effects of a method such that it adapts well to the various calling context. For the analysis to be compositional, we cannot by design rely on much information from specific call sites.

1.1 Contributions

This thesis makes the following contributions:

- We present an inter-procedural effect and alias analysis for the Scala programing language. Our analysis works on arbitrary Scala code, assuming the absence of concurrency and provided that the complete source code is available. Our analysis builds on previous work on inter-procedural pointer analysis for Java [Sal06]. We adapted and extended the precision and scope of the original technique with the following features:
  - A differentiation between strong and weak field updates to detect definitely destructive assignments.
  - A refinement of the allocation site abstraction that summarizes sets of object; by incorporating part of the call-stack information in the labelling of allocation sites, we increase the precision of the analysis for some common patterns, such as factory methods.
  - A recency abstraction to be able to determine when an allocation site abstracts a unique object (singletons).
- We have implemented our analysis in a tool called Insane whose source code is freely available.[1] Insane extends the official Scala compiler and can thus in principle be used with any Scala program.

[1] `http://github.com/colder/insane`

1.2 Outline

The rest of this thesis is organized as follows: Chapter 2 gives a quick overview of the tool, followed by an in-depth description of the initial analysis phases. Chapter 3 describes in full details the pointer and effect analysis phase. In Chapter 4, we describe implementation details. Chapter 5 describes previous work done in the field of pointer and effect analysis. We then conclude in Chapter 6 with some ideas for future work.

Chapter 2

Tool Overview

The tool is made of five distinct phases that are topologically ordered in terms of dependencies. We briefly describe each of the phases necessary for the overall analysis. We then describe each major phases in its respective section. The tool is decomposed as follows, in order:

1. Function Extraction: collects function definitions seen in the abstract syntax tree provided by the Scala compiler and extracts pre- and post-conditions.

2. Control Flow Graph Generation: generates a control flow graph for each function definition, composed of simple assign statements.

3. Class Hierarchy Analysis: collects information to find all subclasses of a class.

4. Type Analysis: analyzes potential runtime types and builds the initial callgraph.

5. Pointer Analysis: analyzes aliasing effects occurring in each function. We describe this phase in full detail in Chapter 3.

2.1 Function Extraction

Each function definition seen in the code is first collected. We start by extracting invariants as well as pre- and post- conditions explicitly stated in the code. We illustrate how Scala allows to express these conditions in Figure 2-1.

```
class A (val next: A) {
  def test(a: A, b: A) = {
    require(a ne b) // pre−condition

    var c = a
    while(c ne b) {
        assert(c ne null) // invariant
        c = c.next
    }
    c

  } ensuring( r ⇒ r eq b ) // post−condition
}
```

Figure 2-1: Expressing invariants and pre-/post-conditions. In Scala, == and ! = are not necessarily checking reference equality. For strict reference equality, Scala defines *eq* and *ne*.

2.2 Control Flow Graph Generation

For each function definition extracted previously, we generate the corresponding control flow graph by converting complex statements into simple assignments. We define the set of simple assignments in Figure 2-2.

Scala converts most operations to method calls, and introduces implicit getters and setters for non-private fields. For example, the expression

```
val a = 2 ∗ this.f
```

is translated by the compiler into

```
val a = 2.∗(this.f())
```

where ∗ is a method on the class Int, and f is the implicit getter method. We then translate it into a simpler form analogous to *three-address code*:

CFG Statement	Code example
AssignCast	`r = v.asInstanceOf[T]`
AssignTypeCheck	`r = v.isInstanceOf[T]`
AssignVal	`r = v`
AssignFieldRead	`r = obj.f`
AssignFieldWrite	`obj.f = v`
AssignNew	`r = new T`
AssignApplyMeth	`r = obj.meth(..args..)`
AssignEQ	`r = v1 eq v2`
AssignNE	`r = v1 ne v2`

Figure 2-2: CFG Statements

val tmp1 = **this**.f()

val tmp2 = 2.∗(tmp1)

val a = tmp2

We also explicitly decompose the instantiation statement $a = new\ \ C(a_1, ..., a_n)$ into two operations: the allocation ($a = new\ \ C$) and the constructor call ($a. < init > (a_1, ..., a_n)$).

When generating the CFG for the main constructor of a given class, we start by assigning the fields defined by the class to their initial value. We assign **null** to fields of non-primitive type, 0 to integer fields, etc.

To illustrate the CFG generation phase, we provide in Figure 2-3 the graph for the method defined in Figure 2-1 without its pre- or post-conditions.

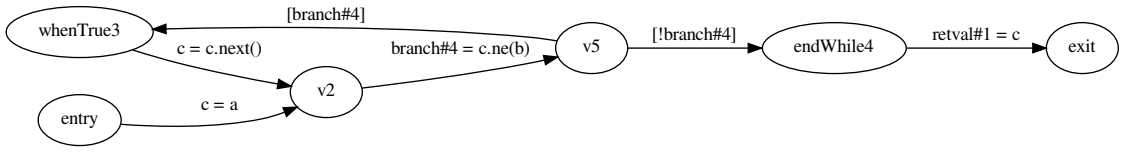


Figure 2-3: CFG Representation

2.3 Class Hierarchy

This phase is responsible for building the complete graph representing the class hierarchy. From this complete class hierarchy, we are able to obtain information such as $subtypes(C) := \{T \mid T, C \in Classes \wedge T \sqsubseteq C\}$, which is required by later phases.

It is worth noting that in object-oriented languages such as Scala, *subtypes*() is generally not bounded. This is due to the fact that most classes are not defined as *final*, so that users can extend and redefine parts of them. Although Scala defines the concept of *sealed* class that forces all *direct* subtypes to be defined in the same source file, this is not sufficient to ensure finiteness of subtypes, because it only bounds direct children. Therefore, children of non-sealed parents can be defined anywhere. In other words, the *sealed* property is not transitive.

To provide a useful analysis, we assume whole-program analysis: we consider that classes analyzed represent the entire program, which allows us to have bounded *subtypes*() sets. We believe that this is a reasonable and pragmatic assumption, as it would not be useful to analyze classes while assuming that most methods could be in fact arbitrarily overridden in imaginary sub-classes.

2.4 Type Analysis

2.4.1 Introduction

Object oriented languages such as Scala implement *dynamic dispatch*: the target of a method call is only determined at runtime, based on the actual runtime type of the receiver. This feature is essential in object oriented languages as it allows subtype polymorphism. Consider the Scala code in Figure 2-4: the declared type of obj in A.test is A, but the target of the method call could either be A.foo or B.foo, based on the actual type of the value of obj, which is only fully determined at runtime.

```
class A {
  def run {
    test(new B)
  }
  def test(obj: A) {
    obj.foo()
  }
  def foo() {
    println("A")
  }
}
```

```
class B extends A {
  override def foo() {
    println("B")
  }
}
```

Figure 2-4: Dynamic dispatch

In general, every redefinitions of foo in all subclasses of A could be targets of this method call. We formalize this concept by associating for each method call a set of targets, CT. For this example, we have:

$$CT(\texttt{obj.foo()}@p) = \{A.foo, B.foo\}$$

where p is the program point–or label uniquely identifying the call.

Type analysis is responsible for computing this set of targets CT. For this analysis to be valid, the set of targets should include all methods that could be called at runtime. It may however be imprecise and include methods that will never be called at runtime.

A simplistic implementation of this analysis would be the following. Consider a call rec.foo() where the receiver rec is of type T, all subtypes of T where method foo() is redefined:

$$SimpleCT(\texttt{rec.foo(..)}@p) := \{C.foo \quad | \quad C \in Classes \wedge C \sqsubseteq type(\texttt{rec}) \wedge \texttt{foo} \in methods(C)\}$$

where $methods(C)$ is the set of methods explicitly (re)defined in class C. This analysis is sound, but it will often be imprecise, as illustrated in Figure 2-5. Even though the type of obj is A, so subtypes are $\{A, B\}$, the only possible target of obj.foo() is A.foo.

```
class A {
    def invoke {
        val obj = new A()
        obj.foo()
    }
    def foo() {
        println("A")
    }
}
```

```
class B extends A {
    override def foo() {
        println("B")
    }
}
```

Figure 2-5: Example of imprecision of the simplistic approach

However, we note that a lack of precision in this analysis will only result in poorer time performance, and will not impact the precision of the overall analysis. Indeed, this analysis is only used to build the initial call graph. Even though the simplistic approach described before might be sufficient in practice, we define a slightly more precise type analysis for this call graph generation. An even more precise type analysis will be performed during the pointer analysis phase.

2.4.2 Analysis

Analyzing the targets of method calls can be reduced to analyzing the runtime types of variables. Those types then fully determines the targets of the call. Our analysis

will thus analyze types that could occur at runtime, for every variable present in the code. We distinguish three types of variables:

1. A.f: Field f of class A.
2. arg: Argument arg of the function.
3. locVar: Local variable locVar.

For each of these variable occurrences in the code, our analysis will compute the set of runtime types, that we will call *ComputedTypes*, as opposed to *RuntimeTypes* which is the set of all types that could occur in runtime. For the resulting type analysis to be valid, the set of computed object types should be a superset of the types of values assigned to those variables at runtime. We thus have the following validity requirement:

$$\forall v \in Variables : RuntimeTypes(v) \subseteq ComputedTypes(v)$$

To compute the set of types used at runtime, we track values assigned to those variables. When doing this, we immediately face two non-trivial problems:

1. The values of arguments are determined by call-sites, determining call-sites of a certain method is analogous to determining call targets, which is the purpose of type analysis.
2. Fields can be assigned from multiple locations, within various methods. Again, determining whether those methods are called, and in which order, requires type analysis.

Both of those problems could be solved using a fix-point mechanism. However, at the cost of some precision, we decided to fall back to a simple implementation for

both arguments and fields:

$$\begin{aligned} ComputedTypes(\texttt{A.f}) &:= \{T \mid T \sqsubseteq type(\texttt{A.f})\} \\ ComputedTypes(\texttt{arg}) &:= \{T \mid T \sqsubseteq type(\texttt{arg})\} \end{aligned}$$

where $type()$ is the statically declared type.

For local variables, we run a flow-sensitive, context-insensitive, abstract interpretation-based analysis. This analysis is thus intra-procedural. It computes, at every program point, the set of all types assigned to local variables. For this analysis to be efficient, we split the type information into two sets T_{sub} and T_{ex}. T_{ex} contains "exact" types, while T_{sub} contains types from which we need to also consider subtypes. This split is useful for two reasons: first, it allows us to keep a small representation for subtypes of, e.g. Object. Second, it lets us delay the resolution of the actual types until the last moment. Globally, storing runtime types with that representation is much more memory efficient than storing a plain set of potential types. We formally define the type information as follows:

$$\begin{aligned} TypeInfo &:= \langle T_{sub} \sqsubseteq Types, T_{ex} \sqsubseteq Types \rangle \\ Types &:= Classes \cup \{Array[T] \mid T \in Types\} \end{aligned}$$

We thus have a point-wise lattice L over pairs of sets of types. Its point-wise lowest upper bound operation is naturally defined as:

$$\langle T_{sub_a}, T_{ex_a} \rangle \sqcup \langle T_{sub_b}, T_{ex_b} \rangle = \langle T_{sub_a} \cup T_{sub_b}, T_{ex_a} \cup T_{ex_b} \rangle$$

We outline in Figure 2-6 the abstraction function for important values.

The number of used types, although infinite in theory, is finite for a given program given that it typechecks. For this reason, we can argue that this analysis terminates,

Expression ex	Abstract Value $\alpha(ex)$
`new A`	$\langle \emptyset, \{A\} \rangle$
`null`	$\langle \emptyset, \emptyset \rangle$
`A.f`	$\langle \{type(\texttt{A.f})\}, \{type(\texttt{A.f})\} \rangle$
`rec.meth(..)`	$\langle \{type(\texttt{rec.meth})\}, \{type(\texttt{rec.meth})\} \rangle$

Figure 2-6: Abstraction function α, where $type()$ returns the declared type.

since there are only finite ascending chains in the lattice as $\mathcal{P}(Types)$ is finite. We also have naturally monotonic transfer functions.

When the fix-point is reached, we can derive the set of call targets CT for each method call using $facts$ computed at their program point:

$$CT(\texttt{rec.meth(..)}@p) := \{T.meth \mid T \in \gamma(facts@p(\texttt{rec})) \wedge \texttt{meth} \in methods(T)\}$$

where γ is the concretisation function, computing the entire set of types that the pair represents:

$$\gamma(\langle T_{sub}, T_{ex} \rangle) := \{T \mid \exists S \in T_{sub}.T \sqsubset S\} \cup T_{ex}$$

This analysis provides us with a relatively precise information on call targets. We believe that the obvious lack of precision in the presence of fields and arguments is not problematic, since the call graph is only used to determine groups of mutually recursive functions. For this reason, this analysis might even be overly precise. However, we have seen that it is sufficiently fast in practice.

Chapter 3

Pointer and Effect Analyses

The problem of analyzing pointers is closely related to the field of effects analysis. Indeed, establishing the relationships between pointers require understanding how and to what values fields are written to. Because of this strong inter-dependence, it is profitable to perform both analyses simultaneously.

Our analysis builds summaries of methods, both in terms of their effects and in terms of the shape of the heap. The analysis stores these summaries as graphs. To compute the graph corresponding to each method, the analysis performs a flow-sensitive abstract interpretation, and computes such a graph at each program point. For method calls, we inline the graph of the callee into the caller's graph. For mutually recursive methods, we iteratively update the graphs until we reach a fix-point. The next section describes how those graphs are generated and what their semantics are. This representation is similar to the graph representation developed in [Sal06]. We can point out few important differences:

- His work is primarily focused on escape analysis while we are interested mostly in effects, and thus we do not retain escape-specific information in our abstract representation.
- We introduce multiple kinds of global nodes that were not present in his work since it would be redundant for escape analysis
- We improved the precision of the graphs by differentiating between strong and

weak updates, by considering as independent object allocations resulting from different method call sequences, and by using recency abstraction.

- Although the representations are similar, the semantics of our graphs as well as the procedures to build them are thus different.

3.1 Graph Semantics

Our representation is based on labelled directed graphs augmented with some meta-data. They are defined by:

$$
\begin{aligned}
G := \ \langle \ & N \subseteq Nodes, \\
& E \subseteq Edges, \\
& locVar \subseteq Variables \times \mathcal{P}(N), \\
& RetNodes \subseteq N \text{ (Nodes representing the return value)} \\
Edges := \ & IEdges \cup OEdges \\
IEdges := \ & \langle N, f, N \rangle \in Nodes \times Fields \times Nodes \text{ (Inside Edges)} \\
OEdges := \ & \langle N, f, N \rangle \in Nodes \times Fields \times Nodes \text{ (Outside Edges)} \\
Variables := \ & \text{All local variables and arguments} \\
Fields := \ & \text{Field names} \\
Nodes := \ & \{GBNode, NNode\} \cup INodes \cup LNodes \\
& \cup PNodes \cup OBNodes \cup LitNodes \\
INodes := \ & \langle T \rangle \in TypeInfo \times ProgramPoints \text{ (Allocation node)} \\
LNodes := \ & \langle from, via, T \rangle \in Nodes \times Fields \times TypeInfo \text{ (Load node)} \\
PNodes := \ & \langle i, T \rangle \in \mathbb{N} \times TypeInfo \text{ (Param node)} \\
OBNodes := \ & \langle T \rangle \in TypeInfo \text{ (Object Node)} \\
LitNodes := \ & \langle T \rangle \in TypeInfo \text{ (Literal values of primitive types)} \\
NNode := \ & \text{(Null node)} \\
GBNode := \ & \text{(Global node)}
\end{aligned}
$$

Nodes generally represent sets of objects (including literals of primitive types), while edges represent may/must-point-to relations. G also contains $locVar$ which is a mapping from local variables to sets of nodes and Ret consists of the set of nodes

that are returned form the procedure.

We present the various abstract representations in a graphical manner. Instead of attaching the metadata on the side, we draw it directly on the graph using the following convention: we draw local variables as border-less nodes, with an edge to each node they point to, drawn in blue. Return nodes are drawn using a double circle. Inside edges are drawn using a full edge, while outside edges are dashed. To illustrate this, we represent in Figure 3-1 the code and graph corresponding to

$$
\begin{aligned}
G &= \langle N, E, locVar, Ret \rangle \text{ with} \\
N &:= \{n_1 : PNode(0), n_2 : LNode(n_1, f2, [60, 26]), n_3 : INode(@[37]), \\
&\quad n_4 : NNull\} \\
E &:= \{IEdge(n_1, f1, n_2), IEdge(n_1, f2, n_3), IEdge(n_3, f1, n_4), \\
&\quad IEdge(n_3, f2, n_4), OEdge(n_1, f2, n_2)\} \\
locVar &:= \{this \mapsto \{n_1\}, tmp \mapsto \{n_2, n_3\}\} \\
Ret &:= \{n_2, n_3\}
\end{aligned}
$$

```
class T(var f1: T, var f2: T) {
  def test() = {
    var tmp = this.f2
    if (a != null) {
      tmp = new T(null, null)
    }
    this.f1 = tmp
    tmp
  }
}
```

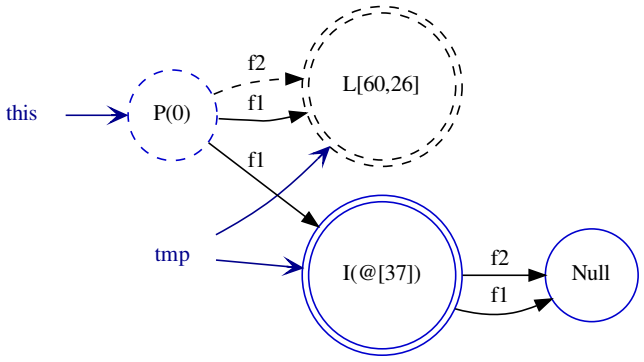


Figure 3-1: Sample code and resulting graph representation

3.1.1 Nodes

As stated earlier, nodes represent sets of objects. Independently of their kind, each node carries two pieces of information:

- The type information *TypeInfo*, corresponding to computed over-approximation of the runtime types of the objects represented by the nodes.
- A *singleton* flag, indicating whether the node represents only one object or may represent more than one. For instance, an allocation in a loop results in multiple objects being represented by the same allocation site[CWZ90].

We now briefly describe each kind of node and their general meaning in a graph. Each kind of node is accompanied with the actual graphical representation, so that nodes are easily recognizable in subsequent examples.

Inside Nodes Inside nodes (INodes) represent objects explicitely allocated via a `new T` statement. The graph thus contains one *INode* per allocation site. The type of this node is exactly the allocated type (`T`).

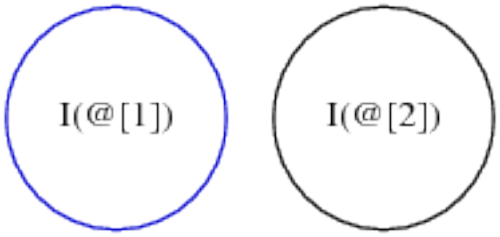


Figure 3-2: Two inside nodes at program points @1 and @2. Blue nodes denote singleton objects.

Load Nodes Intuitively, load nodes (LNodes) represent objects that are not yet determined. For instance, the code:

```
def foo(arg: A) {
    val a = arg.f
    a
}
```

generates a load node representing the objects that arg.f points to at the time of the call to foo(). Section 3.5.5 describes in full details how we resolve load nodes when inlining the graph of the callee into the caller. Note that we do not introduce load nodes unless necessary. For example, consider the following code:

```
def foo(arg: A) {
    arg.f = arg
    val a = arg.f
    a
}
```

we know that arg.f points to arg at the time of the read, and thus we do not need to introduce a load node. Load nodes are conservatively assuming that they may represent many objects, and their attached type is the declared type of the field and all its subtypes.

Figure 3-3: Load Nodes are always dashed. Numbers used in the representation are used to uniquely identify them. Load nodes are never singletons.

Parameter Nodes Parameter nodes (PNodes) represent arguments of the current procedure. Parameter nodes are indexed by the position of the argument. The object corresponding to the current instance (**this**) is implicitly defined as the *PNode* of index 0. For example, the following function definition:

```
class A {
    def foo(arg1: B, arg2: C) // ...
}
```

yields three parameter nodes: $PNode(0)$ of type A and subtypes, $PNode(1)$ of type B and subtypes and $PNode(2)$ of type C and subtypes. Parameter nodes are always

considered singletons.

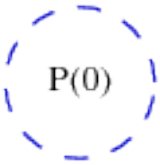


Figure 3-4: Representation of a parameter node. They are always dashed and considered as singletons.

Object Nodes Scala provides an elegant way of defining singletons using the **object** keyword. Object nodes (OBNodes) represent each of those global singleton objects. Naturally, they represent a single object, and their type information is exactly the type of the singleton.

Figure 3-5: Representation of a object node.

Global Node The global node (GBNode) represents all possible nodes, it is thus of type *Object* and subtypes, and is not singleton. This node is only used in rare cases, for instance to represent errors: an inside edge to this global node is similar to a *havoc* operation on the corresponding field.

Figure 3-6: Representation of a global node, of all types and never singleton.

Literal Nodes Literal Nodes represent the literal values used within the code. Except for Strings, those values are not in fact objects. They thus do not have any outgoing edge, and are here solely to record effects of non-object fields. As one would expect, literal nodes hold the type of the literal. They are also represented as singletons; even though they conceptually represent several values, these nodes never have outgoing edges and therefore regarding them as a unique value is not a problem.

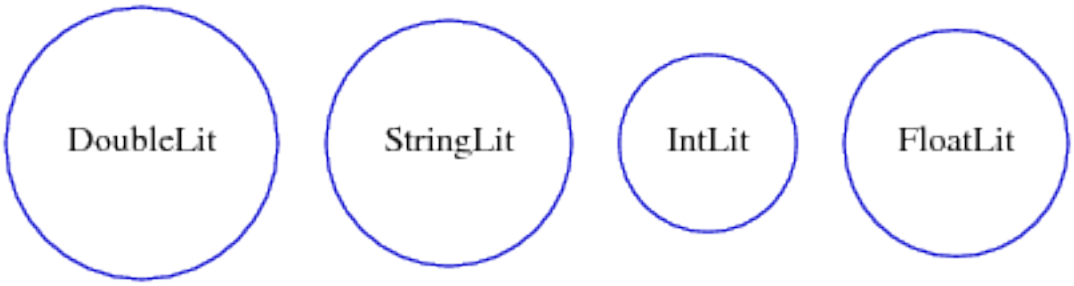


Figure 3-7: Representation of some literal nodes.

Null Node The null node (NNode) represents the empty set of objects corresponding to the *null* value. It holds no type, and is considered singleton.

Figure 3-8: Representation of a null node.

3.1.2 Edges

We distinguish between two kinds of edges, *inside* and *outside* edges. We now briefly describe the meaning of each:

Inside Edges Inside edges represent write operations to fields. Inside edges are drawn with a continuous arrow, labelled by the field on which the write occurs. More than one inside edge with the same field can originate from the same node, signaling that multiple values could be assigned to the field.

Outside Edges The intuition behind outside edges is that they encode a way to reach load nodes. In other words, they represent a read on a field value that does not have any corresponding node yet. Outside edges are thus closely related to load nodes. In fact, it is a property of our graphs that every load node is reachable from a non-load node by following outside edges only. Outside edges are drawn as dashed edges, labelled by the field read.

To illustrate both kinds of edges, we consider in Figure 3-9 an example of conditional update, along with the relevant parts of the resulting graph. At the end of

```
class Plop(var next: Plop) {
  def test(a: Plop) = {
    if (a != null) {
      a.next = a
    }
  }
}
```

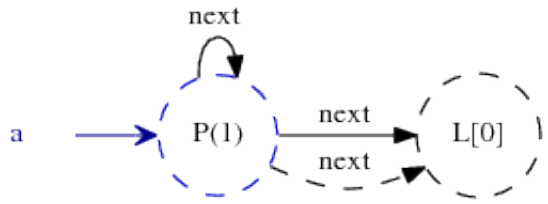


Figure 3-9: Example of weak update in code and corresponding graph.

the test method, a.next either points to a or to the old value, represented by the load node, which depends on the actual object passed to test().

Now that graphs have been described, we provide the definitions of some utility functions along with notations that will be used throughout this report.

$$
\begin{aligned}
nodes(G)(v \in Variable) &:= G.locVar(v) \\
types(node \in Nodes) &:= node.types \text{ (Type information attached to the node)} \\
singleton(node \in Nodes) &:= node.isSingleton \text{ (Whether the node represents one object)} \\
types(G)(v \in Variable) &:= \bigcup \{types(node) | node \in nodes(G)(v)\} \\
e.v1 &:= \text{The source node of the edge e} \\
e.f &:= \text{The field labelling the edge e} \\
e.v2 &:= \text{The destination node of the edge e}
\end{aligned}
$$

We omit specifying the graph G whenever it is clear from the context.

3.2 Weak and Strong Updates

The concept of weak or strong updates relates to the possibility for an update to discard old values. For example, after the execution of the following code:

```
def test() = {
    val a = new Counter
    a.value = 2
    a.value = 3
}
```

the expected value of a.value is 3, and never 2. In this case, the updates are *strong*: they discard previously assigned values. However, in the following code:

```
def test() = {
    val a1 = new A
    val a2 = new A
    val a = if (..) a1 else a2
    a.value = 2
}
```

we know that a points to either a1 or a2. It would however be wrong to assume that a1.value or a2.value is now exclusively 2. Generally speaking, we allow a strong update for $r.f = v$ whenever r represents a single object. In more formal terms, we have that the condition for a strong update on $r.f = v$ is:

$$(|nodes(r)| = 1) \wedge (\forall n \in nodes(r).\ singleton(n))$$

We also have to consider strong updates performed in a branch. Consider the following code:

```
def test() = {
    val a = new A
```

```
    a.f = 1
    if (..) {
        a.f = 2
        a.f = 3
    }
}
```

Even though some updates are conditional, we should be able to infer that after the branch, the value of a.f is either 2 or 3, but never 1. We describe how branches are handled with respect to strong/weak updates in Section 3.3, describing the lattice and its join operation.

In case of a weak update, we can no longer discard the old value, and we still need to have an inside edge pointing to it. In case it is not yet determined by the code, we introduce a load node that represents this previous value. When the graph is later inlined, this load node is then mapped to the actual old value, as described in Section 3.5.5. Figure 3-9 represents such a case.

In order to perform a precise alias analysis, it is key to be able to detect strong updates as often as the code permits, as discarding old values obviously improves the overall precision of the analysis. A very important example is field initialization: without strong updates, every object field would be treated as potentially pointing to null, which is the default value directly after the allocation of the object.

3.3 Lattice Definition

The lattice that we use in our effect analysis contains the graphs described previously as elements. As the graph encompasses information about all (relevant) variables, there is no need to consider a point-wise extension of the lattice.

3.3.1 Join Operation

We now describe the least upper bound ($\sqcup$) operation, also commonly referred to as *join*. We generalize it to take an arbitrary number of arguments $args := \{a_1, ...a_n\}$.

Intuitively, it is the union of all graphs, with one exception regarding inside edges: if one node present in all branches has an inside edge of a field originating from it in only some of the branches, we have to explicitly introduce a load node along with its corresponding inside and outside edges. In other words, if one branch performs a write on a field that is left untouched in other branches, we have to act as if these branches wrote to the field its own previous value. We define the *join* operation in Algorithm 1.

Algorithm 1 Lattice Join Operation

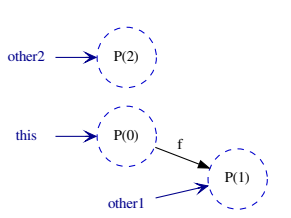
1: **function** $\bigsqcup(graphs = \{G_1, .., G_n\})$
2: **if** $|graphs| = 1$ **then**
3: **return** x s.t. $x \in graphs$
4: **else**
5: $N_{common} \leftarrow \bigcap_i N_i$
6: $Pairs_{all} \leftarrow \bigcup_i \{\langle ie.v1, ie.f\rangle \mid ie \in G_i.E \wedge ie \text{ is IEdge}\}$
7: $Pairs_{common} \leftarrow \bigcap_i \{\langle ie.v1, ie.f\rangle \mid ie \in G_i.E \wedge ie \text{ is IEdge}\}$
8: $N_{load} \leftarrow \{safeLNode(p.v1, p.f, @0) \mid p \in Pairs_{all} - Pairs_{common} \wedge p.v1 \in N_{common}\}$
9: $E_{load} \leftarrow \{IEdge(in.v1, in.f, in) \mid in \in N_{load}\} \cup \{OEdge(in.v1, in.f, in) \mid in \in N_{load}\}$
10: **return** $\langle \bigcup_i G_i.N \cup N_{load}, \bigcup_i G_i.E \cup E_{load}, \bigcup_i G_i.locVar, \bigcup_i G_i.R\rangle$
11: **end if**
12: **end function**

We now consider three code examples illustrating the different cases. Each time, the graph of both branches are provided, along with the graph resulting from joining them.

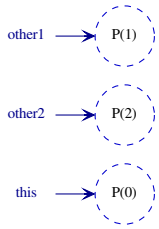
Example 1 Figure 3-10 contains a conditional update, without any previous reference to the conditionally written field. This represents the corner case in which we need to introduce a load node: indeed, we have $N_{common} = \{P(0), P(1), P(2)\}$, $allPairs = \{\langle P(0), f\rangle\}$, and $Pairs_{common} = \emptyset$.

Example 2 Figure 3-11 contains a conditional update, preceded by a write on the same field. In this case, no load node needs to be introduced.

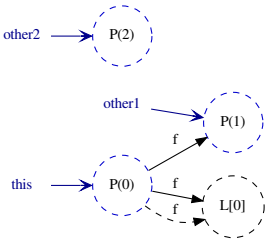
```
class A {
  var f: A = null
  def test(other1: A, other2: A) {
    if ( .. ) {
      this.f = other1 // Branch 1
    } else {
                        // Branch 2
    }
                        // Result
  }
}
```



(a) Branch 1



(b) Branch 2

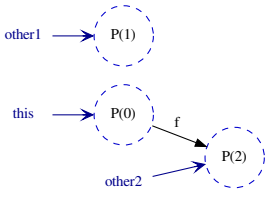


(c) Result

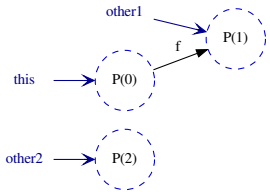
Figure 3-10: Example 1: introduction of a load node

Example 3 Figure 3-12 illustrate the case of a conditional write on the same field with the same *source* node, but with distinct destinations. We are able to establish that after this conditional update, $this.f$ no longer retains its old value. This code is equivalent to the previous example, and indeed yields the same effects.

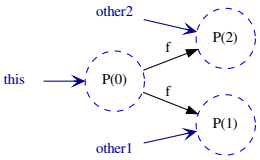
```
class A {
  var f: A = null
  def test(other1: A, other2: A) {
    this.f = other1
    if ( .. ) {
      this.f = other2 // Branch 1
    } else {
                    // Branch 2
    }
                    // Result
  }
}
```



(a) Branch 1



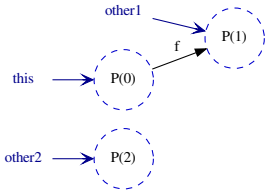
(b) Branch 2



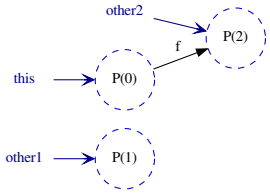
(c) Result

Figure 3-11: Example 2: Strong update followed by a conditional update

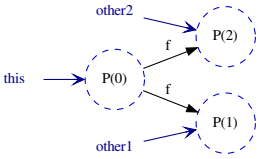
```
class A {
  var f: A = null
  def test(other1: A, other2: A) {
    if ( .. ) {
      this.f = other1 // Branch 1
    } else {
      this.f = other2 // Branch 2
    }
                    // Result
  }
}
```



(a) Branch 1



(b) Branch 2



(c) Result

Figure 3-12: Example 3: two conditional strong updates

3.4 Graph-based Type Analysis

Even though our global type analysis was flow sensitive, it suffered from critical imprecision with respect to fields and arguments. For this reason, we implemented a type analysis based on our graphs. The approach we took is to attach type information to each node. Since nodes represent objects, this is a sensible thing to do. The additional information we store alongside each node is similar to what we used in our previous type analysis: $\langle T_{sub}, T_{ex} \rangle$ a pair of two sets of types T_{sub} and T_{ex} where T_{sub} represents the set of types from which we need to include subtypes, and T_{ex}. The set of runtime types attached to each node is determined based on the node types. Figure 3-13 illustrates the main cases. We then use those types in order to compute the set of potential targets for a method call. Given the call `obj.foo()`, we obtain the set of runtime types corresponding to `obj` as follows:

$$types(\texttt{obj}) = \bigcup \{\gamma(types(n)) \mid n \in nodes(\texttt{obj})\}$$

we can then look for potential targets in the resulting set of types, similarly to what we did in our previous type analysis.

Node Type	Types Associated
INode(A)	$\langle \emptyset, \{A\} \rangle$
LNode(a, f)	$\langle \{type(a.f)\}, \{type(a.f)\} \rangle$
PNode(arg)	$\langle \{type(arg)\}, \{type(arg)\} \rangle$
OBNode(A)	$\langle \emptyset, \{A\} \rangle$
NNode	$\langle \emptyset, \emptyset \rangle$
GBNode	$\langle \{Object\}, \{Object\} \rangle$ (all)
Literal Nodes	$\langle \emptyset, \{type(Literal)\} \rangle$

Figure 3-13: Summary: types associated to each kind of nodes.

If we pessimistically consider that each field read yields a *load node*, this type analysis is exactly as precise as the one described previously. The improvements come from two sides:

- In practice, not every field read yield a load node, for instance, a read performed after a strong update only targets the newly assigned value, which might be of

a more precise set of types.

- When inlining the graph of of method call, parameter nodes are mapped to other nodes at the call site, and load nodes are resolved, if possible. The natural inlining of methods makes this type analysis context sensitive.

We consider in Figure 3-14 an example illustrating this improvement in precision.

```
class A {
  var f: A = null
  def setF(a: A) { f = a }

  def test(obj: A) {
    val a = new A
    a.setF(a)
    a.f.foo()
  }
  def foo() {
    println("A")
  }
}
```

```
class B extends A {
  override def foo() {
    println("B")
  }
}
```

Figure 3-14: Improved graph-based type analysis

At the time of the call to a.f.foo() we have from the graph at that program point that a.f is the *inside node* corresponding to the object from **new** A. We thus obtain $\langle\{\}, \{A\}\rangle$ for a.f instead of $\langle\{A\}, \{A\}\rangle$, which excludes $B.foo$ from the call.

3.5 Transfer Functions

We now describe in Figure 3-15 the transfer functions illustrating the transformations to the environment caused by most relevant statements. For complex operations, we describe their effect in a specific function.

3.5.1 Allocations

The $alloc(Graph, r, C, @p)$ operation is responsible for creating the appropriate inside node for the class C and assign it to r. $@p$ represents the program point at which

Statement st	Transfer Function f
`r = v`	$\langle N, E, locVar[r \mapsto nodes(v)], R\rangle$
`r = new C` @p	$alloc(G, r, C, @p)$
`r = null`	$\langle N \cup \{NNode\}, E, locVar[r \mapsto \{NNode\}], R\rangle$
`r.f = v` @p	$write(G, nodes(r), f, nodes(v), @p)$
`r = v.f` @p	$read(G, nodes(v), f, r, @p)$
`r = v.meth(a1, .., an)` @p	$call(G, r, v, meth, (a_1, ..., a_n))$
`return v`	$\langle N, E, locVar, nodes(v)\rangle$

Figure 3-15: Transfer function f

the allocation occurs, in other words it represents the allocation site.

During allocation, we also need to determine the *singleton* flag of the corresponding INode. There are two ways of generating inside nodes corresponding to multiple objects: loops and function calls. We provide here the general idea that is able to handle both cases soundly. A refinement of this technique in the presence of function calls is described in Section 3.5.5.

Algorithm 2 describes the transformations made to the graph by *alloc*. The idea is simple: before actually including the corresponding inside node into the environment, we look whether it is already present. In such case, we are in the presence of a loop, and this inside node needs to be replaced by a *non-singleton* inside node. Otherwise, we add the inside node with *singleton* set to *true*. Figure 3-16 provides an example of cycle detection.

Algorithm 2 Allocations

1: **function** ALLOC($\langle N, E, locVar, R\rangle, r, C, @p$)
2: $n \leftarrow INode(C, false, @p)$
3: $n_{sgt} \leftarrow INode(C, true, @p)$
4: **if** $n_{sgt} \in N$ **then**
5: $N_{new} \leftarrow (N \cup \{n\}) - \{n_{sgt}\}$
6: $locVar_{new} \leftarrow \{v \mapsto v_{nodes}[n_{sgt} \mapsto n] \mid (v \mapsto v_{nodes}) \in locVar\}$
7: $locVar_{new} \leftarrow locVar_{new}[r \mapsto \{n\}]$
8: **else**
9: $N_{new} \leftarrow N \cup \{n_{sgt}\}$
10: $locVar_{new} \leftarrow locVar[r \mapsto \{n_{sgt}\}]$
11: **end if**
12: **return** $\langle N_{new}, E, locVar_{new}, R\rangle$
13: **end function**

```
class A() {
  def test() = {
    var o: A = null
    while (o == null) {
      o = new A // @p
    }
  }
}
```

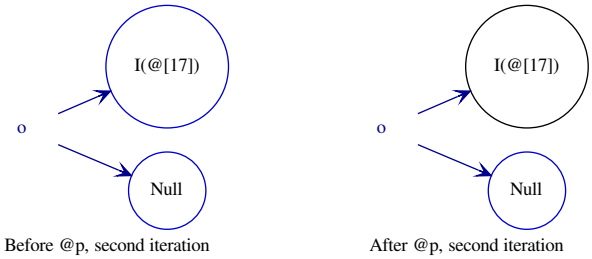


Figure 3-16: Cycle Detection, the inside node goes from *singleton* (blue), to *non-singleton* (black). Graphs represent the state during the second loop iteration, before and after @p.

3.5.2 Field Updates

The $write(Graph, from, f, to, @p, allowStrong)$ operation is responsible for managing field updates. It represents the modifications done by writing the nodes *to* to the field f of nodes $from$, at program point @p. The $allowStrong$ argument specifies whether strong updates can be performed by this write operation. The use for this argument will only become apparent when we describe the mechanics used for method calls.

Algorithm 3 describes the required graph transformations. The main idea is that, in case of a strong update, we simply remove old inside edges and add the new one. The rule is more complex in the case of weak updates , because we need to keep the old value as well. To determine the old value, we start by following inside edges. If none exists, we follow outside edges. If this old value is still not determined at this point, we introduce a load node to represent it.

Algorithm 3 Field Updates

1: **function** WRITE($\langle N, E, locVar, R\rangle, from, f, to, @p, allowStrong$)
2: $isStrong \leftarrow \forall n \in from.\ n.isSingleton \wedge |from| = 1 \wedge allowStrong$
3: $N_{new} \leftarrow N$
4: **if** $isStrong$ **then**
5: $E_{new} \leftarrow E - \{ie \mid ie \in E \wedge ie$ is IEdge $\wedge\ ie.v1 \in from \wedge ie.f = f\}$
6: $E_{new} \leftarrow E_{new} \cup \{IEdge(v_{from}, f, v_{to}) \mid v_{from} \in from \wedge v_{to} \in to\}$
7: **else**
8: **for** $n_{from} \leftarrow from$ **do**
9: $previous \leftarrow \{ie.v2 \mid ie \in E \wedge ie$ is IEdge $\wedge\ ie.v1 = n_{from} \wedge ie.f = f\}$
10: $E_{new} \leftarrow E$
11: **if** $previous = \emptyset$ **then**
12: $previous \leftarrow \{ie.v2 \mid oe \in E \wedge ie$ is OEdge $\wedge\ oe.v1 = n_{from} \wedge oe.f = f\}$
13: **end if**
14: **if** $previous = \emptyset$ **then**
15: $lNode \leftarrow safeLNode(n_{from}, f, @p)$
16: $E_{new} \leftarrow E_{new} \cup \{IEdge(n_{from}, f, lNode), OEdge(n_{from}, f, lNode)\}$
17: $N_{new} \leftarrow N_{new} \cup \{lNode\}$
18: **end if**
19: $E_{new} \leftarrow E_{new} \cup \{IEdge(n_{from}, f, v_{to}) \mid v_{to} \in (previous \cup to)\}$
20: **end for**
21: **end if**
22: **return** $\langle N_{new}, E_{new}, locVar, R\rangle$
23: **end function**

```
class A(var f: A) {
  def test(a1: A) = {
    val a2 = new A(null)
    val r = if (a1 != null) a1 else a2
    r.f = this
  }
}
```

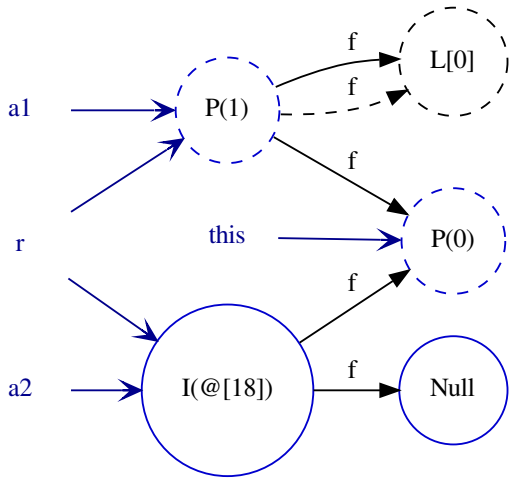


Figure 3-17: Example of a weak update, where one of the old values is undetermined.

3.5.3 Field Reads

The $read(Graph, to, f, from, @p)$ operation is responsible for managing field reads. It represents the modifications done by reading the field f from the nodes $from$ and assigning the result to to, at program point $@p$.

Algorithm 4 describes the transformations done by *read*. Intuitively, *read* tries to determine a previous value by first following inside and then outside edges. If no such value can be found, it introduces a load node representing this value. We illustrate this using an example in Figure 3-18 displaying both scenarios.

Algorithm 4 Field Reads

```
1:  function READ(⟨N, E, locVar, R⟩, from, f, to, @p)
2:      N_new ← N
3:      E_new ← E
4:      pointed ← ∅
5:      for n_from ← from do
6:          previous ← {ie.v2 | ie ∈ E ∧ ie is IEdge ∧ ie.v1 = n_from ∧ ie.f = f}
7:          if previous = ∅ then
8:              previous ← {ie.v2 | oe ∈ E ∧ ie is OEdge ∧ oe.v1 = n_from ∧ oe.f = f}
9:          end if
10:         if previous = ∅ then
11:             lNode ← safeLNode(n_from, f, @p)
12:             E_new ← E_new ∪ {OEdge(n_from, f, lNode)}
13:             N_new ← N_new ∪ {lNode}
14:             pointed ← pointed ∪ {lNode}
15:         else
16:             pointed ← pointed ∪ previous
17:         end if
18:     end for
19:     return ⟨N_new, E_new, locVar[to ↦ pointed, R⟩
20: end function
```

```
class A(var f: A) {
  def test(a: A) = {
    val r1 = a.f
    a.f = this
    val r2 = a.f
  }
}
```

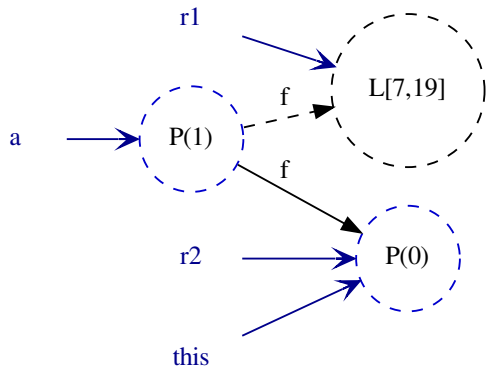


Figure 3-18: Example of read on a field. At the time of $r1 = a.f$ no previous value is found: a load node is introduced. For $r2 = a.f$ the previous value is found by following inside edges first.

3.5.4 Creation of Load Nodes

In the previous algorithms, we used a $safeLNode$ function that is responsible for *safely* creating new load nodes in a way that guarantees the analysis will terminate. For example, if $safeLNode(from, field, @p)$ were to naively create a new $LNode(from, field, @p)$, the analysis would not necessarily terminate!

```
class A (next: A) {
  def traverse = {
    val current = this
    while(current != null) {
      current = current.next // @p
    }
  }
}
```

Figure 3-19: Problematic code with respect to load nodes

We now walk through our analysis to illustrate why it would not terminate with such a definition of $safeLNode$, given the code provided in Figure 3-19:

1. After the first iteration, the read has created the load node $l_1 := LNode(P(0), next, @p)$, since $P(0).next$ was yet undetermined.

2. At the beginning of the second iteration, we have $nodes(current) = \{P(0), l_1\}$.

3. After the second iteration, the read has created the load node $l_2 := LNode(l_1, next, @p)$, since $l_1.next$ or $P(0).next.next$ was undetermined.

4. And so on, a new load node is created at each iteration, the analysis is thus unable to terminate.

In order to solve this issue, we need to limit the way we generate new load nodes, so that their number is always bounded. Algorithm 5 describes how we define $safeLNode$ for such purpose.

Algorithm 5 Safely Creating Load Nodes

```
1: function SAFELNODE(from, f, @p)
2:     if from is LNode then
3:         from_new ← from.from
4:     else
5:         from_new ← from
6:     end if
7:     return LNode(from_new, f, @p)
8: end function
```

3.5.5 Method Calls

For method calls, the effects of the method are represented by the graph resulting from the analysis of the function. The callgraph computed after type analysis allows us to group inter-dependent functions together (in strongly connected components). We can then perform a topological sort of those strongly connected components, which basically orders the analysis in such a way that most methods will already have been analyzed when encountering one of their call sites.

```
class A {
  var f: A = null
  def test(a1: A, a2: A, a3: A) {
    a2.f = this.f
    a1.f = a2
    a1.f = a3
  }
  def plop() {
    val o1 = new A
    val o2 = new A
    val o3 = if (this!=null) o1 else o2
    o1.test(o3, o2, o2)
  }
}
```

Figure 3-20: Code example used to illustrate inlining mechanics

To illustrate how methods are handled, we provide a code example containing a method call in Figure 3-20 in which various operations are performed. We describe in Algorithm 7 how we handle statements representing method calls. Given the statement $r = v.meth(a_1, .., a_n)$, we first need to resolve the types corresponding to v, so that we can collect all methods *meth* that might be the targets of this call. We then *inline* the graph for each target separately, and then join the resulting environments.

The formal description of the inlining algorithm is rather complex, but its intuition is relatively easy to understand: we start by mapping each write operations from the

Algorithm 6 Method Call

1: **function** CALL($G, r, v, meth, (arg_1, ..., arg_n), @p$)
2: $types_{rec} \leftarrow resolve(types(v))$
3: $methods \leftarrow \{m \mid T \in types_{rec} \wedge m \in methods(T) \wedge m.name = meth.name\}$
4: $nodes_{rec} \leftarrow nodes(v)$
5: $nodes_{args} \leftarrow (r_1, ..., r_n)$ s.t. $r_i = nodes(arg_i)$
6: **return** $\bigsqcup$ { $inlineGraph(G, m, Graphs(m), nodes_{rec}, nodes_{args}, @p) \mid m \in methods$}
7: **end function**

Algorithm 7 Graph Inlining

1: **function** INLINEGRAPH($G_{caller}, meth, G_{callee}, recNodes, argsNodes, @p$)
2: $\langle map, G_{new} \rangle \leftarrow buildMap(G_{caller}, meth, G_{callee}, recNodes, argsNodes, @p)$
3: **repeat**
4: $G_{old} \leftarrow G_{new}$
5: $mapEdges \leftarrow \{\langle IEdge(mv_1, e.f, mv_2), e.v_1 \rangle \mid e \in G_{callee}.E \wedge e$ is IEdge $\wedge mv_1 \in map(e.v_1) \wedge mv_2 \in map(e.v_2)\}$
6: $sources \leftarrow \{\langle e.v_1, e.f \rangle \mid \langle e, o \rangle \in mapEdges\}$
7: **for** $\langle v, f \rangle \leftarrow sources$ **do**
8: $edges \leftarrow \{e \mid \langle e, o \rangle \in mapEdges \wedge \langle e.v_1, e.f \rangle = \langle v, f \rangle\}$
9: $olds \leftarrow \{o \mid \langle e, o \rangle \in mapEdges \wedge \langle e.v_1, e.f \rangle = \langle v, f \rangle\}$
10: $allowStrong \leftarrow \forall o \in olds \; . \; |map(o)| = 1$
11: $G_{new} \leftarrow write(G_{new}, \{v\}, f, \{e.v_2 \mid e \in edges\}, @p, allowStrong)$
12: **end for**
13: **until** $G_{new} = G_{old}$
14: **end function**

callee to corresponding write operations in the caller, using the previously computed *map*. Once we have the corresponding write operations grouped by the node and field they affect, it remains to determine whether this write operation could be strong. Given a write on $v.f$, a strong update is allowed if v was only pointed to from the callee by nodes with v as unique target in the map. In other words, if

$$\forall v_{orig} \in map^{-1}(v) \ . \ map(v_{orig}) = \{v\}$$

It is worth noting that allowing a strong update at this stage does not necessarily mean that a strong update is indeed performed. It is only enabling the *write* operation to perform one if it sees fit to do so. In our example, none of the performed writes end up being strong updates.

Mapping Nodes Between Graphs

For this inlining to be fully defined, it remains to describe how this *map* is initially computed. Algorithm 8 describes how this mapping is performed. It naturally starts by mapping nodes corresponding to the parameters of the function, including the 0th argument representing the receiver. All global nodes are also mapped to themselves.

For inside nodes, we first precise the allocation site by composing it with the point at which the method is called, we do it in a way such that no repetitions are allowed. Otherwise, it would not terminate in the case of a recursive function. We also, like in the case of an allocation statement, figure out this node's *singleton* flag, by detecting loops. In case we previously had a *singleton* node and detected a cycle, we replace it with a *non-singleton* node and adjust the map accordingly. We display in Figure 3-21 the mapping relation between the two graphs at the point of the method call in Figure 3-20.

In the presence of *load nodes*, we need to see if they are fully determined in the caller graph, so that we can resolve them. The resolution is made using their *from* node along with their field. Since this node could itself be a yet unresolved load node, we first make sure that it is resolved by calling $resolveLoad$ on it. As in the case of a

Algorithm 8 Building Node Map

1: **function** BUILDMAP($G_{caller}, meth, G_{callee}, recNodes, argsNodes, @p$)
2: $G_{new} \leftarrow G_{caller}$
3: $map \leftarrow \{P(0) \mapsto recNodes\}$
4: **for** $i \leftarrow 1$ *to* $|meth.args|$ **do**
5: $map = map \cup \{P(i) \mapsto argsNodes_i\}$
6: **end for**
7: **for** $n \in LitNodes \cup OBNodes \cup \{NNode, GBNode\}$ **do**
8: $map = map \cup \{n \mapsto \{n\}\}$
9: **end for**
10: **for** $in \in \{n \mid n \in G_{callee}.N \wedge n \text{ is INode}\}$ **do**
11: $pPoint \leftarrow safeCompose(n.pPoint, @p)$
12: $iNode_{sgt} \leftarrow INode(types(in), true, pPoint)$
13: $iNode \leftarrow INode(types(in), false, pPoint)$
14: **if** $iNode_{sgt} \in G_{new}.N \vee iNode \in G_{new}.N$ **then**
15: $G_{new}.N \leftarrow (G_{new}.N \cup \{iNode\}) - \{INode_{sgt}\}$
16: $map \leftarrow map[iNode_{sgt} \mapsto iNode] \cup \{in \mapsto \{iNode\}\}$
17: **else**
18: $G_{new}.N \leftarrow G_{new}.N \cup \{iNode_{sgt}\}$
19: $map \leftarrow map \cup \{in \mapsto \{iNode_{sgt}\}\}$
20: **end if**
21: **end for**
22: **for** $ln \in \{n \mid n \in G_{callee}.N \wedge n \text{ is LNode}\}$ **do**
23: $\langle map, G_{new} \rangle = resolveLoad(ln, map, G_{new}, @p)$
24: **end for**
25: **return** $\langle map, G_{new} \rangle$
26: **end function**

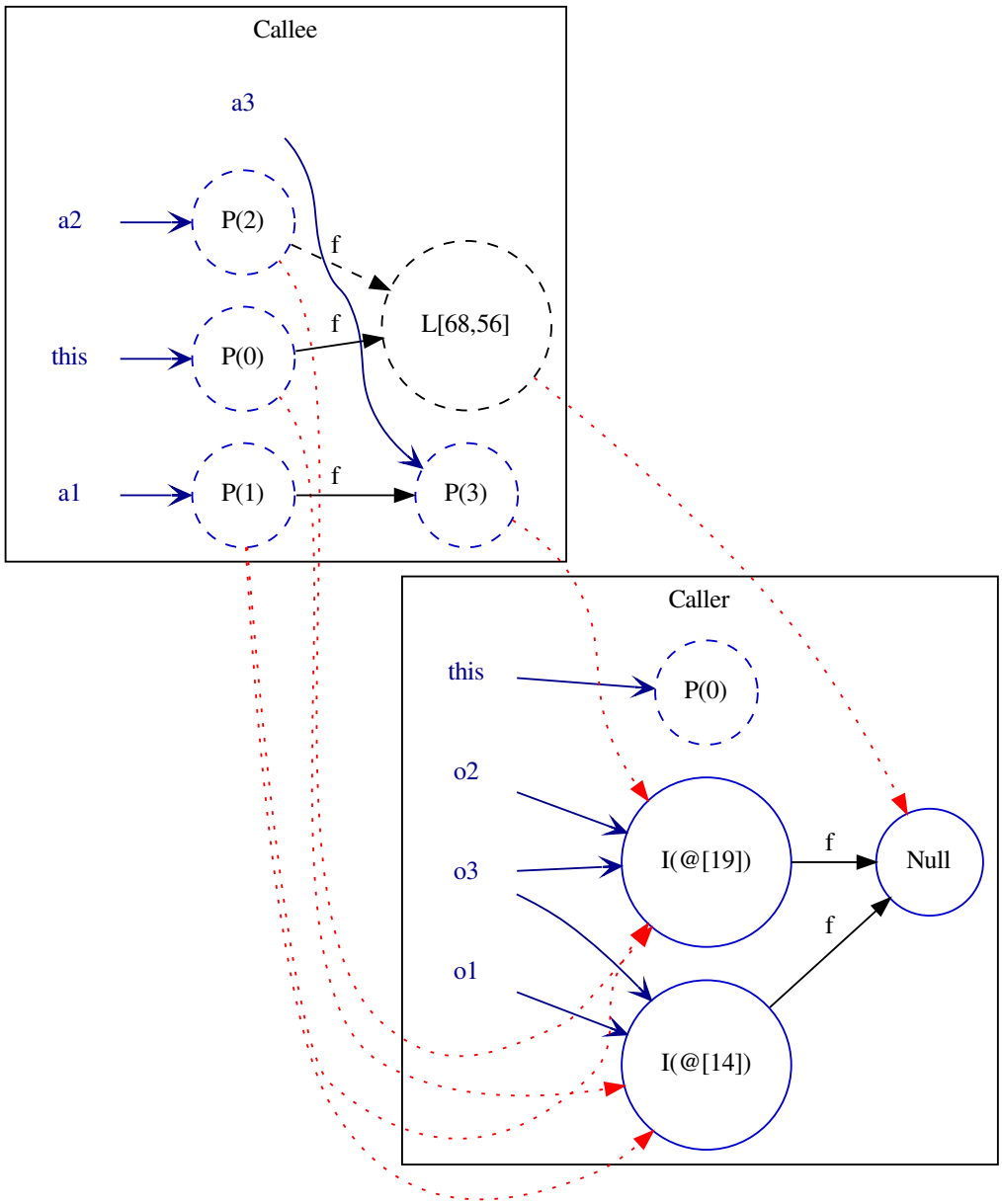


Figure 3-21: Map relation between G_{callee} and G_{caller} for the method call in Figure 3-20. The map is displayed in dotted red arrows.

read operation, we follow first inside and then outside edges. If no target is found, we need to maintain the load node in the resulting graph. Here again, we safely refine its program point. In our example, the load node is determined by following inside edges from $I([@19])$ via f, we obtain the set of nodes $\{NNull\}$.

Algorithm 9 Resolving Load Nodes

```
1:  function resolveLoad(lNode, map, G, @p)
2:      LNode(from, f, pPoint) ← lNode
3:      if from is LNode then
4:          ⟨map, G⟩ = resolveLoad(from, map, G, @p)
5:      end if
6:      for n ∈ map(from) do
7:          targets ← {ie.v2 | ie ∈ G.E ∧ ie is IEdge ∧ ie.v1 = n ∧ ie.f = f}
8:          if targets = ∅ then
9:              targets ← {ie.v2 | oe ∈ G.E ∧ ie is OEdge ∧ oe.v1 = n ∧ oe.f = f}
10:         end if
11:         if targets = ∅ then
12:             pPoint ← safeCompose(pPoint, @p)
13:             newLNode ← safeLNode(from, f, pPoint)
14:             G.E ← G.E ∪ {IEdge(n, f, newLNode), OEdge(n, f, newLNode)}
15:             G.N ← G.N ∪ {newLNode}
16:             map ← map ∪ {lNode ↦ {newLNode}}
17:         else
18:             map ← map ∪ {lNode ↦ targets}
19:         end if
20:     end for
21:     return ⟨map, G⟩
22: end function
```

To conclude, we provide in Figure 3-22 the graph resulting from the complete inlining of the method call from our example.

3.5.6 Analyzing Inter-dependant and Recursive Methods

As explained earlier, we analyze groups of inter-dependent functions together. They are represented by strongly connected components in our call-graph. We construct the effects of the methods iteratively, assuming that methods have initially no effect (i.e. their effect graph is empty). The algorithm for the analysis of strongly connected components is outlined in Algorithm 10 and consists of a standard fix-point

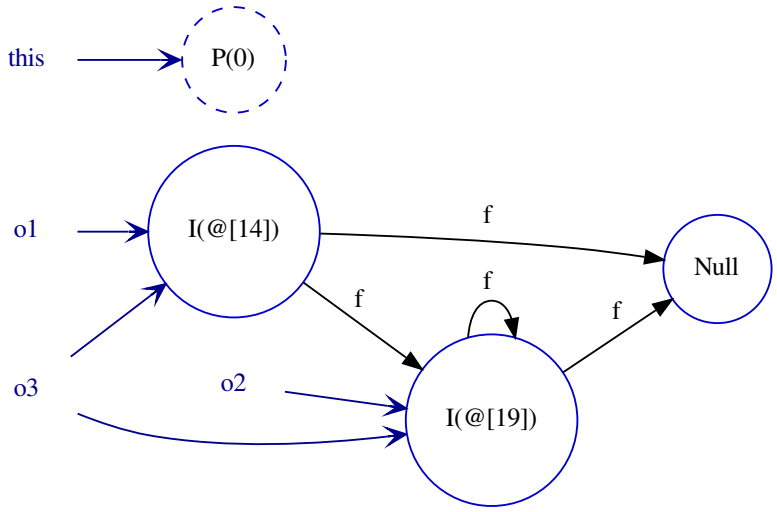


Figure 3-22: Resulting graph after the $o1.test(..)$ method call in Figure 3-20.

mechanism.

Algorithm 10 Analyzing Mutually Recursive Functions

```
1:  function ANALYZESCC(scc : {m_1, ..., m_n})
2:      workList ← scc
3:      while |workList| > 0 do
4:          m s.t. m ∈ workList
5:          workList ← workList − m
6:          G_before ← Graphs(m)
7:          analyze(m)
8:          G_after ← Graphs(m)
9:          if G_before ≠ G_after then
10:             workList ← workList ∪ {t | t calls m }
11:         end if
12:     end while
13: end function
```

3.6 Argument for Termination

We have seen during the description of the parts composing the analysis that in order to ensure termination, specific care needs to be taken. We provide here an informal

argument showing analysis termination.

First of all, we argue that the graphs have a limited number of nodes. We have two types of nodes that are created dynamically: inside nodes and load nodes. Inside nodes are specific to certain program points. The number of basic program points is bounded by the size code. It remains to show that the $safeCompose$ operation only generates a bounded number of composed program points. The function prevents any repetitions of program points, we thus have that $safeCompose$ can only generate a finite number of composed program points. Then, we consider load nodes, which are determined by a node, a field and a program point. The number of fields are of course bounded. Without load nodes, the number of nodes is bounded. However, we must prevent an infinite chain of load nodes. Our $safeLNode$ specifically prevents that, and thus only allows a finite number of load nodes to be generated. Thus, the number of nodes is finite. Given that edges are only determined by two nodes and a field, there is also only a finite number of them. In the end, we thus have a finite number of distinct graphs.

The argument above is however not sufficient to claim termination, as we still have to show that our transfer function is monotonic. Again, we argue informally that it is, given that the only operation reducing the number of edges is a strong update. However, it is never the case that on a given pair of graphs G_1, G_2 s.t. $G_1 \sqsubset G_2$, that some strong update is applicable in G_2 but not in G_1. A more in-depth demonstration, especially with respect to graph inlining, would be required to produce a formal proof, but is beyond the scope of this work.

3.7 Loop/Function Equivalence

Even though Scala can be used as a one-to-one substitute for an imperative language, its features favor functional programming. It is thus critical to provide a precise inter-procedural analysis. We demonstrate in this section that recursive loops and their corresponding functional version are equivalently represented as graphs. For instance, Figure 3-23 provides two equivalent implementations of the same code, one

imperative and one functional.

```
val c: Constants = initConstants
var s: State = initState
while(cond(s, c)) {
    s = update(s, c)
}
s
```

```
def f(s : State, c : Constants) : State = {
 if(cond(s, c)) {
   f(update(s, c), c)
 } else {
   s
 }
}
f(initState, initConstants)
```

Figure 3-23: Two equivalent implementations, for arbitrary functions *cond* and *update*

We expect from the effect graphs to be the same. Indeed, it would be profitable for two main reasons:

- Scala translates certain call patterns into while loop for efficiency reasons. And thus, the inference of the effects of a while loop should be well handled.
- Since Scala encourages functional programming, a tool targeted at Scala programs is expected to be precise in presence of functional code, and notably recursive function calls.

We argue informally by comparing the fix-point technique used in a loop and the fix-point performed within a strongly connected component in the call-graph (mutually recursive functions). First, let us introduce some recursive $fixpoint$ function defined as:

$$\begin{aligned} fixpoint(init, \mathcal{F})(0) &:= init \\ fixpoint(init, \mathcal{F})(n) &:= fixpoint(init, \mathcal{F})(n-1) \cup \mathcal{F}(fixpoint(init, \mathcal{F})(n-1)) \end{aligned}$$

Loop Fix-Point We start by assuming that the effects of the entire looping block (in this case the effects of `s = update(s, c)`) are compacted into one specific transfer function $\mathcal{F}$, mapping the state at the entry of the loop to the state at the end. Given a state S before the loop, we obtain S after 0 iterations, and $S \cup \mathcal{F}(S)$ after one

iteration. After two iterations, we obtain $S \cup \mathcal{F}(S) \cup \mathcal{F}(S \cup \mathcal{F}(S))$ which corresponds to $fixpoint(S, \mathcal{F})(2)$. Naturally, after n iterations we get $fixpoint(S, \mathcal{F})(n)$.

Function Fix-Point When handling mutually recursive functions, we start by assuming functions have no effect. We then analyze each function until the graph of every functions in the strongly connected component are stable. In the present case, we start by analyzing f assuming that f has yet no effect. The first call to f produces thus no effect.

At the end of this first analysis, we obtain the initial state S which is given by the first parameter. The second analysis thus considers a function returning its first argument as effect, and thus the effect of the call is again $\mathcal{F}(S)$. Joined with the other effect-less branch, we obtain $S \cup \mathcal{F}(S)$. A subsequent iteration yields: $S \cup \mathcal{F}(S) \cup \mathcal{F}(S \cup \mathcal{F}(S)) = fixpoint(S, \mathcal{F})(2)$. After n recursive call, we again obtain $fixpoint(S, \mathcal{F})(n)$. Provided that $\mathcal{F}$ is monotonic (which is the case given that it is a composition of calls to our transfer function), and that our lattice does not allow infinite ascending chains, we have that $fixpoint(S, \mathcal{F})(\infty)$ exists.

We have thus informally established that both fix-points represent the same effects. And thus, our analysis has the same precision for either programming style.

3.8 Subsequent Analyses

Using the information gathered by the alias and effect analysis previously described, we can perform subsequent analyses. We briefly describe three of them:

3.8.1 Purity Analysis

Using the effects graph computed for each method, we can easily determine if a certain function has no *observable* effects. We define an observable memory effect as a write operation on parameter, object, or global nodes. It naturally follows that a method is considered *observably pure* if no inside edge can be reached, while traversing the graph starting from parameter, object, or global nodes. This can be checked using a

standard depth-first-search algorithm.

3.8.2 Nullness Analysis

Because we represent the null value as a special node, we can detect whether a variable or object field is potentially *null*. We are thus able to emit warnings in case a read or write operation is performed from a potentially *null* value.

3.8.3 Object Initializations

Checking whether object fields are correctly initialized before usage is a special case of nullness analysis. Even if a field has a default value, this value will only be assigned depending on the initialization order. Using the field prior to that point may thus yield a NullPointer exception. Scala allows mixins-based compositions of classes using *traits*, which complicates the initialisation phase considerably. However, because the Scala compiler linearizes the traits in a phase preceding ours, we are able to analyze the initialization by following a simpler chain of super classes, similar to what we would find in Java.

Chapter 4

Implementation

We have implemented the techniques presented in the previous chapters as part of a tool called Insane. Insane is built as a plugin for the official Scala compiler. As an immediate advantage, it grants us immediate access to the trees and all type information that we need. The compiler allows us to plug our tool between two existing phases. Depending on where the plugin is inserted, it dramatically changes the aspect of the trees. In this case, we decided to put it late in the compilation process, so that we would not have to deal with closures, inner classes, mixins, or genericity. This however comes with a cost: not all information on parametric types is available (because of type erasure), and the amount of code to analyze is much larger.

4.1 Class Hierarchy

The first implementation problem we faced while working with the Scala compiler is that it does not provide any way to access subtypes of one symbol, only its super type. For this reason, we had to traverse every symbol defined in the classpath in order to reconstruct the entire hierarchy, which allowed us to compute subtypes. This process of traversing every defined symbols in the class path is costly; a minimum of 30'000 symbols are always present, given that the Java and Scala library are always included. As a consequence, it dominated the analysis time for small examples.

For this reason, we decided to store the class hierarchy of the Scala and Java libraries into a database. We use a nested set[1] representation for our hierarchical data, which allows us to retrieve the entire set of subtypes in one SQL query efficiently.

4.2 Pointer and Effect Analysis

As we have seen in Chapter 3, our analysis computes effects graphs for each method present in the analyzed source. When calling a method, we inline the graph corresponding to the target method into the caller graph.

4.2.1 Library Dependencies

One of the major problems that we faced while trying to analyze even small Scala programs is that references to the Scala library are ubiquitous even if they are not explicitly stated in the source. This is caused by the various compiler phases prior to ours, that expand high-level construct into combinations of lower-level ones. To illustrate this issue, we consider an apparently self-contained example in Figure 4-1, and the corresponding code at the time of our phase in Figure 4-2. We can see that even though the original code made no explicit references to the Scala library, the actual code that we analyze does.

```
class A (val next: A*)

object Test {
  def test() = {
    val a = new A()
    val b = new A(a)
  }
}
```

Figure 4-1: Self-contained example in Scala

[1] `http://dev.mysql.com/tech-resources/articles/hierarchical-data.html`

```
package <empty> {
  class A extends java.lang.Object with ScalaObject {
    private[this] val next: Seq = _
    def next(): Seq = A.this.next
    def this(next: Seq): A = {
      A.this.next = next
      A.super.this()
      ()
    }
  }
  final object Test extends java.lang.Object with ScalaObject {
    def test(): Unit = {
      val a: A = new A(immutable.this.Nil)
      val b: A = new A(scala.this.Predef.wrapRefArray(
            Array[A]{a}.asInstanceOf[Array[java.lang.Object]]()))
      ()
    }
    def this(): object Test = {
      Test.super.this()
      ()
    }
  }
}
```

Figure 4-2: Corresponding code at our compiler phase

4.2.2 Storing Intermediate Results

Those ubiquitous references to the Scala library force us to analyze the library along with the code. Given that the library consists of approximately 90'000 methods, it would be very inefficient to require the library code to be included alongside each piece of code we want to analyze.

Thankfully, the modularity of the graph-based effects representation allows us to pre-calculate the graphs for every method of the library and store them in a database. To store those results in the database, we had to implement a special serialization procedure, as the state stored in the graphs contained references to internal compiler structures that were not serializable. Our custom serialization allows for full recovery of the state.

It is worth noting that this pre-calculation is only possible under the assumption that the effects of library methods are "self-contained". This is better expressed in

terms of the shape of the call-graph: there should be no calls from the library to non-library methods. This assumption generally does not hold, in the presence of higher-order functions, or callbacks. This is covered in more details in Section **??**.

4.2.3 Unanalyzable Methods

While analyzing the library, we stumbled upon the fact that it is itself not self-contained but heavily references the Java library. In overall, it calls over 700 distinct methods from the Java library.

Even though we could in theory apply the same analysis to the Java source code, our analysis was implemented on top of the Scala compiler, which is unable to compile Java source code. We thus were not able to apply the same pre-calculation technique used for the Scala library directly.

Instead of making conservative assumptions and applying *havoc* on every object involved in those various library calls, we provided a way for the user to provide Scala stub ("dummy") implementations of those Java classes and methods. This is achieved via annotations put at the level of the classes and methods, that informs our compiler that this graph is meant to represent a different methods. We illustrate this use of annotations in Figure 4-3 with an excerpt of the java.math.BigInteger Scala implementation.

```
package insane
package predefined

import annotations._

@AbstractsClass("java.math.BigInteger")
class javamathBigInteger {
  @AbstractsMethod("java.math.BigInteger.abs(()java.math.BigInteger)")
  def abs(): java.math.BigInteger = {
    new java.math.BigInteger("42")
  }

  // ...

  @AbstractsMethod("java.math.BigInteger.valueOf"+
                   "((x1: Long)java.math.BigInteger)")
  def valueOf(x1: Long): java.math.BigInteger = {
    new java.math.BigInteger(x1)
  }
}
```

Figure 4-3: Dummy Scala implementation of java.math.BigInteger using annotations

Chapter 5

Related Work

To our knowledge, we are the first trying to perform such alias analysis for Scala. Although Scala compiles to Java bytecode, and thus any analyzer working with bytecode could in principle be used for analyzing Scala, the steps performed during compilation introduce many artifacts. For that reason, an analysis focused on Scala will be able to provide much more useful and precise result than one working with arbitrary Java bytecode.

Given the usefulness of alias analysis, it has been constantly worked on in the past decades and remains an active area. Most of the time, alias analysis is not the goal but the mean to achieve a more sophisticated analysis.

The work that is naturally the most related to this thesis is the work done by Alexandru Salcianu in [Sal01, Sal06], on which this thesis builds. They provide a compositional graph based pointer analysis that focus on establishing escaping information. While we also provide a similar compositional analysis based on graphs, we assign slightly different semantics to our graphs to cope with strong updates, that they did not support. We also handle program points refinement, which allows us to provide a more precise analysis in the presence of factory methods.

In [DDA10b], they propose to add invariants to refine pointer relations in the heap. Our analysis is thus less precise in that regard, as it is currently not path sensitive at all. In [CJ07], they develop a demand-driven alias analysis. It is however flow insensitive and thus less precise that what we have here. However, the fact that

it is demand-driven is very interesting.

In [DMM96, BS96, JS01], they describe similar type analysis techniques to compute the call graph in the presence of dynamic dispatch.

One distinction between our work and [Sal06] is that we differentiate between weak and strong updates. Researchers have looks at ways to further refine this paradigm by introducing logical predicates that qualify the uniqueness of the object summary [DDA10a]; although the technique appears appealing, we do not expect that it would provide much benefit in our setting. Indeed, our analysis already treats most updates as strong, and it is not clear whether parametrizing them with predicates would allow us to perform a strong update when we previously could not.

Different alias analysis techniques have been explored in order to perform type state analysis [SY86, SY93]. In [FD02], instead of figuring out heap aliasing, required for type state checking, they propose a type system extension that inherently restricts aliasing interferences. In [HO10], they design an annotation system to describe messages passed between concurrent objects so that they can be used without any risk of race-conditions.

Much work has been done in order to obtain guarantees during object initialization [QM09, FX07, CJ07]. For instance, [FX07] proposes type-based techniques to prevent issues from arising during the initialization phase. In our analysis, we handle initialization in a straightforward manner by inlining the graphs from the various constructors. This allows us to detect whether at some point during or after initialization, an object field retain its default value (i.e. null).

Constructing a precise call-graph in the presence of higher order functions has been known to be a problematic analysis. It has been shown that it is in fact equivalent to dynamic dispatch analysis [MSH10]. In Scala, the link between the two is explicit, since closures get compiled into classes defining an apply method. We could thus benefit from ideas developed by [Shi88] and later refined by others to improve our dynamic dispatch analysis for calls to closures.

Chapter 6

Limitations and Future Work

We conclude with a discussion of some of the current limitations of Insane, as well as directions for future improvements.

6.1 Concurrency

Considering arbitrary interleaving would considerably complicate our analysis. Indeed, the concepts of previous value or strong update are not intuitively defined in the presence of concurrency. For this reason, we decided to ignore concurrency. Since Scala promotes an actor model for implementing concurrent programs, we believe that it is reasonable to consider a sequential execution of the code. This would be safe if objects passed as messages are not arbitrarily modified on both ends, which is a property that we could check using our analysis, but currently do not.

6.2 Exceptions

It is worth noting that we currently do not handle exceptions. In effect, we ignore exception handlers (e.g. **try**/**catch** blocks), and assume that **throw** statements redirect the flow directly to the end of the procedure. This way of handling is not only precise, but it is also not valid in theory. However, even though we think that this should be fixed in future versions, we believe that improving the handling of exceptions should

only have a limited impact on the resulting precision in terms of effects and alias information.

6.3 Nullness Analysis

Given that our analysis performs strong updates whenever possible, we are able to obtain a relatively precise *nullness* analysis for free. Indeed, if during a call $r = obj.foo$ or a field access $obj.f$, we have that $NNode \in nodes(obj)$, we can raise a warning about a potential null dereference that could cause the program to crash entirely. However, this remains a rather primitive analysis, as it currently does not take branching conditions into account. As a result, even the following code would generate a spurious warning:

```
if (a != null) {
    a.foo()
}
```

It would be interesting to improve the precision of this analysis in order to potentially spot bugs in existing Scala applications.

It is worth noting that Scala implicitly discourages the use of *null* to indicate the absence of values. Rather, it defines an *Option[T]* datastructure with two subtypes $Some[T](val)$ or $None$. However, nothing prevents a developer to blindly call Option's *get* method, which in the case of $None$ is equivalent to a null-dereference. We could certainly apply the same principles here in order to detect such cases.

6.4 Higher Order Functions

The presence of Higher Order Functions (HOF) complicates our analysis and compromise its precision. In Scala, HOFs are represented as objects, instances of FunctionX classes where X is the number of arguments the function has. To illustrate, we consider a use of a HOF:

```
def test() = {
```

```
    plop(42, _ + 1)
}
def plop(i: Int, f: Int ⇒ Int): Int = f(i)
```

We have here a function named plop which applies the function f passed as second argument to its first argument. In test we call that method with 42, and the function incrementing its argument by one. The result of test should thus be 43. At our phase, the compiler already translated that code to:

```
def test(): Int = {
    plop(42, (new Testanonfun$test$1(): Function1))
}
def plop(i: Int, f: Function1): Int = f.apply$mcII$sp(i);

class Testanonfun$test$1 extends
  scala.runtime.AbstractFunction1$mcII$sp with Serializable {

  final def apply(x1 : $Int$) : $Int = Testanonfun$test1.$this.apply$mcII$sp($x1);

  def apply$mcII$sp(v1: Int): Int =
    v1.+(1);

  final def apply(v1: java.lang.Object): java.lang.Object =
    scala.Int.box(
      Testanonfun$test$1.this.apply(scala.Int.unbox(v1))
    );
}
```

As we can see, the compiler transformed the type Int ⇒ Int into the general type Function1. It also transformed the closure _ + 1 into a class defining, among other things, a apply$mcII$sp method which is the specialized name of the method for Int ⇒ Int. The call to f is transformed into a method call to that apply$mcII$sp.

If we recall how type analysis is performed for arguments of methods, we can immediately see a problem in the presence of HOFs. Indeed, the method plop takes an argument f of type Function1 which is the super type of all functions of one argument. The runtime types calculated for f thus includes *all* closures of one argument, including ones with incompatible types. As a result, any call using f as a receiver potentially targets every defined closures.

In order to address that issue, we could implement the following three techniques:

6.4.1 Exploring Type History

The main reason why types are generalized is that our analysis runs after the *errasure* phase, which is responsible of removing type information that cannot remain at runtime because of JVM limitations (mostly generic types). Thankfully, the compiler keeps an history of the types associated to each symbol. We could thus recollect the type of the arguments prior to the *errasure* phase, allowing us to limit the targets to methods of compatible type. Our recent experience with the compiler tells us that even though that idea is conceptually simple and feasible in theory, there are probably many hurdles to avoid until we obtain a reliable mechanism for recovering this type information.

6.4.2 Selective Analysis Inlining

Even though the previous technique would help eliminate many spurious targets, it would remain highly imprecise. Figure 6-1 provides an example illustrating the imprecision. Assuming that the closures defined in test1 and test2 are the only instance of Function1, the effects inferred for plop is the combination of the effects of the two closures. As a result, we infer that test1 writes to the field a even though it does not.

By selectively inlining the analysis of the method plop in test1 and test2, we could refine the type of the argument from $Function1$ to the exact type of the class generated for each closure. Type analysis would then naturally infer that the f.apply$mcII$sp call in plop has only one target. As a result, the effects of test1 and test2 would be

```
class Test {
    var a: Int = 42

    def test1() = {
        plop(_ + 1)
    }

    def test2() = {
        plop(x ⇒ a += 1; a)
    }

    def plop(f: Int ⇒ Int) = f(42)
}
```

Figure 6-1: Imprecise effects inference

inferred with respect to the closure they define and use.

6.4.3 Graph-based Delaying of Method Calls

Instead of inlining the entire analysis of a method, we explored ways to generate graphs in which certain calls would remain unresolved. The main idea is that in the presence of an imprecise function call, we could replace the call by a special node indicating a method call, and "wait" until the receiver gets refined to actually apply the method call. This refinement would be done automatically by node remapping, given that we use a graph-based type analysis. We could thus keep the overall modularity of our analysis, and decide to delay problematic method calls, which would include but not be limited to HOFs.

Even though this idea is appealing, it is yet still unclear how to manage those delayed calls while preserving soundness. The main difficulties that such a technique would bring are of temporal nature. Assumptions of "happened before" no longer hold in the presence of delayed calls, causing precision loss:

1. Strong updates could no longer be applied at various places, i.e. in the delayed method call.

2. Load nodes that were related to those delayed calls cannot be resolved before

the call itself.

Due to time restrictions, we could not go further with the development of this promising feature.

6.5 Conclusion

We presented Insane, an interprocedural pointer and effect analysis for Scala. Even though the analysis considers the entire program, it is compositional and allows intermediate results to be stored for efficient re-use. While information reuse is currently limited to summaries computed for the Scala standard library, we are let to believe that modifications required for it to be fully incremental would be small, given that it is already compositional. Overall, Insane promises to provide an efficient and precise tool for pointer and effect analysis, an important basic block enabling richer analysis for Scala in the future.

References

[BS96] David F. Bacon and Peter F. Sweeney. Fast static analysis of c++ virtual function calls. In *OOPSLA*, pages 324–341, 1996. 64

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977. 10

[CC02] Patrick Cousot and Radhia Cousot. Systematic design of program transformation frameworks by abstract interpretation. In *POPL*, pages 178–190, 2002. 10

[CJ07] Patrice Chalin and Perry R. James. Non-null references by default in java: Alleviating the nullity annotation burden. In Erik Ernst, editor, *ECOOP*, volume 4609 of *Lecture Notes in Computer Science*, pages 227–247. Springer, 2007. 63, 64

[CWZ90] David R. Chase, Mark N. Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *PLDI*, pages 296–310, 1990. 27

[DDA10a] Isil Dillig, Thomas Dillig, and Alex Aiken. Fluid updates: Beyond strong vs. weak updates. In Andrew D. Gordon, editor, *ESOP*, volume 6012 of *Lecture Notes in Computer Science*, pages 246–266. Springer, 2010. 64

[DDA10b] Isil Dillig, Thomas Dillig, and Alex Aiken. Symbolic heap abstraction with demand-driven axiomatization of memory invariants. In William R. Cook,

Siobhán Clarke, and Martin C. Rinard, editors, *OOPSLA*, pages 397–410. ACM, 2010. 63

[DMM96] Amer Diwan, J. Eliot B. Moss, and Kathryn S. McKinley. Simple and effective analysis of statically typed object-oriented programs. In *OOPSLA*, pages 292–305, 1996. 64

[FD02] Manuel Fähndrich and Robert DeLine. Adoption and focus: Practical linear types for imperative programming. In *PLDI*, pages 13–24, 2002. 64

[FX07] Manuel Fähndrich and Songtao Xia. Establishing object invariants with delayed types. In Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr., editors, *OOPSLA*, pages 337–350. ACM, 2007. 64

[FYD+08] Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. *ACM Trans. Softw. Eng. Methodol.*, 17(2), 2008. 9

[HO10] Philipp Haller and Martin Odersky. Capabilities for uniqueness and borrowing. In Theo D'Hondt, editor, *ECOOP*, volume 6183 of *Lecture Notes in Computer Science*, pages 354–378. Springer, 2010. 64

[JS01] Thomas P. Jensen and Fausto Spoto. Class analysis of object-oriented programs through abstract interpretation. In Furio Honsell and Marino Miculan, editors, *FoSSaCS*, volume 2030 of *Lecture Notes in Computer Science*, pages 261–275. Springer, 2001. 64

[MSH10] Matthew Might, Yannis Smaragdakis, and David Van Horn. Resolving and exploiting the -cfa paradox: illuminating functional vs. object-oriented program analysis. In Benjamin G. Zorn and Alexander Aiken, editors, *PLDI*, pages 305–315. ACM, 2010. 64

[QM09] Xin Qi and Andrew C. Myers. Masked types for sound object initialization.

In Zhong Shao and Benjamin C. Pierce, editors, *POPL*, pages 53–65. ACM, 2009. 64

[Sal01] Alexandru D. Salcianu. Pointer analysis and its applications to Java programs. Master's thesis, MIT, 2001. 63

[Sal06] Alexandru D. Salcianu. *Pointer Analysis for Java Programs: Novel Techniques and Applications*. PhD thesis, MIT, 2006. 10, 23, 63, 64

[Shi88] Olin Shivers. Control-flow analysis in scheme. In *PLDI*, pages 164–174, 1988. 64

[SY86] Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Software Eng.*, 12(1):157–171, 1986. 64

[SY93] Robert E. Strom and Daniel M. Yellin. Extending typestate checking using conditional liveness analysis. *IEEE Trans. Software Eng.*, 19(5):478–485, 1993. 64