Foundations and Trends® in Programming Languages
Vol. 2, No. 1 (2015) 1–69

DOI: 10.1561/2500000014

now
the essence of knowledge

Pointer Analysis

Yannis Smaragdakis
University of Athens
smaragd@di.uoa.gr

George Balatsouras
University of Athens
gbalats@di.uoa.gr

Contents

Abstract

Pointer analysis is a fundamental static program analysis, with a rich literature and wide applications. The goal of pointer analysis is to compute an approximation of the set of program objects that a pointer variable or expression can refer to.

We present an introduction and survey of pointer analysis techniques, with an emphasis on distilling the essence of common analysis algorithms. To this end, we focus on a declarative presentation of a common core of pointer analyses: algorithms are modeled as configurable, yet easy-to-follow, logical specifications. The specifications serve as a starting point for a broader discussion of the literature, as independent threads spun from the declarative model.

Y. Smaragdakis and G. Balatsouras. *Pointer Analysis*. Foundations and Trends® in Programming Languages, vol. 2, no. 1, pp. 1–69, 2015.

DOI: 10.1561/2500000014.

1

Introduction

Pointer analysis or *points-to analysis* is a static program analysis that determines information on the values of pointer variables or expressions. Such information offers a static model of a program's heap. Since the heap is the primary structure for global program data, pointer analysis forms the substrate of most inter-procedural static analyses. Virtually all interesting questions one may want to ask of a program will eventually need to query the possible values of a pointer expression, or its relationship to other pointer expressions. The exact representation of such information, i.e., the static abstraction used to model the heap, often serves as a classifier of the analysis algorithm. Although the literature is not entirely consistent on high-level terminology, pointer analysis is a near-synonym of *alias analysis*. Whereas, however, pointer/points-to analysis typically tries to model heap objects and asks "what objects can a variable point to?", alias analysis algorithms focus on the closely related question of "can a pair of variables or expressions be aliases, i.e., point to the same object?" [Landi and Ryder, 1992, Emami et al., 1994]

In this monograph, we attempt to survey the most common modern approaches to pointer analysis, with an eye towards ease of exposition

and concreteness. Our presentation aspires to be rather more tutorial and hands-on than other surveys of the pointer analysis area. For a thorough view of the literature, with an emphasis on coverage, readers can consult Hind [2001], Ryder [2003], Sridharan et al. [2013], and Kanvar and Khedker [2014].

Our tutorial will develop, in significant detail, a modular, configurable model of a standard points-to analysis. This analysis model is a skeleton on which we progressively add more flesh, to reflect several realistic features and analysis enhancements from the recent literature. The analysis model will also serve as a firm basis for high-level tangential discussions on topics that deviate from the model: alias analysis, complexity theory results, algorithms not captured well by the formal model, and more.

Importantly, our analysis model is executable: The specification of our algorithms is given in Datalog, which is simultaneously a logic and a realistic programming language. The use of Datalog allows us to express the *precision* aspects of pointer analyses concisely, at almost the same high level as a mathematical formalism, yet with no need to separately treat the topic of how to implement the algorithms so that they perform *efficiently*.

The axes of *precision* and *performance/efficiency* characterize every approach to program reasoning. All interesting questions about universal (i.e., all-inputs) program behavior are undecidable—see Landi [1992], Ramalingam [1994], Reps [2000] specifically for pointer analysis problems. Thus, every technique is evaluated both on its precision, i.e., the degree to which the result approximates the uncomputable mathematical ideal, and on its performance, i.e., the asymptotic complexity or practical speed of computation.

We can use these axes to guide a more general overview of the landscape of techniques for reasoning about program memory, of which pointer analysis is only one part. Further along the precision axis lie several approaches such as *shape analysis* [Sagiv et al., 2002] and *separation logic* [Reynolds, 2002, O'Hearn et al., 2001]. Separation logic is a full-fledged logic, typically deep in the forests of undecidability, where reasoning requires close human guidance. Shape analysis is a

computable, automated program analysis, yet with performance complexity in the territory of intractability: complexity bounds for the best-known shape analyses are super-exponential.

In contrast, the term "pointer analysis" is typically reserved for techniques of modest performance cost, scaling to realistic, automated whole-program analysis efforts. This bias in favor of automation and scalability to full realistic programs is also reflected throughout our discussion and analysis formulations. Datalog (with the standard enhancement of an order relation) is a language that captures the $\mathcal{PTIME}$ complexity class [Immerman, 1999, Ch.14]: every Datalog program runs in polynomial time, and every polynomial algorithm can be written in Datalog.

Although a polynomial complexity bound is hardly a guarantee of practical scalability, in broad mathematical strokes it is a reliable distingushing feature. Indeed, it is tempting to consider "pointer analysis" to refer precisely to heap analysis algorithms of polynomial complexity. In our formulation of analyses, we strive to maintain this complexity boundary. This is reflected in our effort to stay within standard Datalog (with stratified negation) and only use extensions as syntactic sugar.

We begin with essential background and an illustration of the best known pointer analysis algorithms.

2

Core Pointer Analysis

The standard formulation of a pointer analysis is as a computation of the set of objects (the *points-to* set) that a program variable may point to during runtime. Let us consider such a computation for a toy example. The example will serve to illustrate multiple points in the remainder of our tutorial.

The program fragment below, in a Java-like language, contains two methods `fun1` and `fun2`. The methods allocate objects and pass them to a third method, `id`, which is the identity function.

```
void fun1() {
  Object a1 = new A1();
  Object b1 = id(a1);
}
void fun2() {
  Object a2 = new A2();
  Object b2 = id(a2);
}
Object id(Object a) { return a; }
```

The first question is how objects are statically abstracted in the analysis. During program run time, methods `fun1` and `fun2` may be invoked millions of times, causing the `new` instructions to allocate an

equal number of distinct objects. Since pointer analysis is done statically, such precision is typically lost in the approximation performed by the analysis. The conventional approach is to represent objects as *allocation sites*, i.e., to consider a single abstract object to stand in for each run-time object allocated by the same instruction.[1] Therefore, in our example, instructions `new A1()` and `new A2()` are each associated with a different abstract object. (Since abstract objects are identified with allocation sites, if another program site contained a textually identical instruction, e.g., `new A1()`, it would still be distinguished from the above. For ease of illustration, we do not show more detailed instruction identifiers and instead opt to keep our examples unambiguous via unique type names in allocation sites.)

Based on this heap abstraction, what would we expect to be the outcome of a pointer analysis of the above program? Certainly variable `a1` in `fun1` (henceforth denoted `fun1::a1`) can point to abstract object `new A1()`, and similarly for `fun2::a2` and `new A2()`. The rest of the variables require inter-procedural reasoning to establish their points-to sets. Since method `id` returns the object passed into it, the result is fairly simple. A fully precise points-to answer for the variables and abstract objects shown is:

`fun1::a1` → `new A1()`
`fun1::b1` → `new A1()`
`fun2::a2` → `new A2()`
`fun2::b2` → `new A2()`
`id::a` → `new A1(), new A2()`

(The → symbol denotes that the left-hand side can point to each of the comma-separated elements of the right-hand side.)

We will next see what actual analysis algorithms compute relative to this precise result.

[1] We will later refine this simple abstraction with the introduction of concepts such as a *context-sensitive heap*, *heap cloning*, the *recency abstraction*, and more. However, the base idea of using allocation sites as identifiers of abstract objects remains in all such refinements. The only significant alternative that we shall examine is the use of *access paths* instead of allocation sites in an alias analysis. Kanvar and Khedker [2014] offer a thorough treatment of the topic of heap abstractions.

2.1 Andersen-Style Points-To Analysis, Declaratively

Perhaps the best-known and most straightforward pointer analysis algorithm family is commonly attributed to Andersen [1994]. An Andersen-style analysis can be easily expressed as *subset constraints*: different program statements induce inferences of the form "points-to set A is a subset of points-to set B", or, computationally, "add all elements from points-to set A to points-to set B".

We shall use the Datalog language to express such constraints/computations. Datalog has been the basis of several implementations of program analyses, both low-level [Reps, 1994, Whaley et al., 2005, Lam et al., 2005, Whaley and Lam, 2004, Bravenboer and Smaragdakis, 2009a, Smaragdakis et al., 2014, Madsen et al., 2013] and high-level [Eichberg et al., 2008, Hajiyev et al., 2006, Naik et al., 2006, 2009].

Computation in Datalog consists of monotonic logical inferences that repeatedly apply to produce more facts until a fixpoint is reached. A Datalog rule "C(z,x) ← A(x,y), B(y,z)." means that if A(x,y) and B(y,z) are both true, for some values x,y,z, then C(z,x) can be inferred. Syntactically, the left arrow symbol (←) separates the inferred facts (i.e., the *head* of the rule) from the previously established facts (i.e., the *body* of the rule). (Pure) Datalog is a close relative of Prolog, but with several significant differences: there are no constructors, thus the domains of computation are always finite, monotonicity is strictly enforced, and there are no computation side-effects.[2] In this way, computation is strictly declarative: the order of evaluation for clauses and rules does not affect the final outcome

For our analysis to be expressed in Datalog, we assume that the program-under-analysis is represented as input relations (typically implemented as database tables) that encode its different elements. Such pre-processing is a relatively straightforward one-to-one translation.

We show the algorithm on a simple intermediate language with a) an "alloc" (or "new") instruction for allocating an object; b) a "move"

[2]In later chapters we shall use an enhanced Datalog notation, with constructors and negation. However these features will be employed only in a disciplined way, as syntactic sugar that can be translated away into pure Datalog relations.

instruction for copying between local variables; c) "store" and "load" instructions for writing to the heap (i.e., to object fields); and d) a "virtual method call" instruction that calls the method of the appropriate signature defined in the dynamic class of the receiver object. This language is a good model for the core features of a multitude of compiler and virtual machine intermediate languages.

Figure 2.1 shows the domain of the analysis (i.e., the different value sets that constitute the space of our computation), its input relations, and the computed (intermediate and output) relations. Our presentation throughout this tutorial will rely on this domain and will incrementally build on the input relations, by mere addition of extra ones. (The computed relations will be revised more drastically for more sophisticated analyses.)

Input relations. The input relations correspond to the intermediate language for our analysis. They are logically grouped into relations that represent instructions and relations that represent name-and-type information. For instance, the ALLOC relation represents every instruction that allocates a new heap object, *heap*, and assigns it to local variable *var* inside method *inMeth*. (Note that every local variable is defined in a unique method, hence the *inMeth* argument is also implied by *var* but is included to simplify later rules.) There are similar input relations for all other instruction types (MOVE, LOAD, STORE, and VCALL).

Similarly, there are relations that encode type system, symbol table, and program environment information. These are mostly straightforward. For instance, FORMALARG shows which variable is a formal argument of a given method at a certain index (i.e., the n-th argument). LOOKUP matches a method signature to the actual method definition inside a type. HEAPTYPE matches an object to its type, i.e., is a function of its first argument. (Note that we are shortening the term "heap object" to just "heap" and represent heap objects as allocation sites throughout.) ACTUALRETURN is also a function of its first argument (a method invocation site) and returns the local variable at the call site that receives the method call's return value. VARTYPE maps a variable to its type and SUBTYPE links a type to its supertypes, while

V is a set of program variables	
H is a set of heap abstractions (i.e., allocation sites)	
M is a set of method identifiers	
S is a set of method signatures (including name, type signature)	
F is a set of fields	
I is a set of instructions	
T is a set of class types	
$\mathbb{N}$ is the set of natural numbers	
ALLOC(*var : V, heap : H, inMeth : M*)	*# var = new ...*
MOVE(*to : V, from : V*)	*# to = from*
LOAD(*to : V, base : V, fld : F*)	*# to = base.fld*
STORE(*base : V, fld : F, from : V*)	*# base.fld = from*
VCALL(*base : V, sig : S, invo : I, inMeth : M*)	*# base.sig(..)*
FORMALARG(*meth : M, n : $\mathbb{N}$, arg : V*)	
ACTUALARG(*invo : I, n : $\mathbb{N}$, arg : V*)	
FORMALRETURN(*meth : M, ret : V*)	
ACTUALRETURN(*invo : I, var : V*)	
THISVAR(*meth : M, this : V*)	
HEAPTYPE(*heap : H, type : T*)	
LOOKUP(*type : T, sig : S, meth : M*)	
VARTYPE(*var: V, type: T*),	
INMETHOD(*instr : I, meth : M*)	
SUBTYPE(*type : T, superT : T*)	
VARPOINTSTO(*var : V, heap : H*)	
CALLGRAPH(*invo : I, meth : M*)	
FLDPOINTSTO(*baseH : H, fld : F, heap : H*)	
INTERPROCASSIGN(*to : V, from : V*)	
REACHABLE(*meth : M*)	

Figure 2.1: Our domain, input relations, and computed relations (for a context-insensitive, Andersen-style analysis). The input relations are of two kinds: relations encoding program instructions (the form of the instruction is shown in a comment) and relations encoding type system and other environment information.

InMethod is a function from instructions to their containing methods. (The latter three relations are not used until Chapter 3.)

Computed relations. There are five output or intermediate computed relations: VarPointsTo, ..., Reachable.[3] The main output relations are VarPointsTo and CallGraph, encoding our points-to and call-graph information—we later discuss the purpose of the latter, together with the analysis specification. The VarPointsTo relation links a variable (*var*) to a heap object (*heap*). Other intermediate relations (FldPointsTo, InterProcAssign, Reachable) correspond to standard concepts and are introduced for conciseness and readability.

Analysis logic. Figure 2.2 shows a complete Andersen-style points-to analysis for our input language. The rules handle all different cases of input statements in the intermediate language. For instance, the first rule handles Alloc statements (i.e., `new` instructions). The rule states that, if we have computed that a method is reachable and it contains the allocation of an abstract object, *heap*, directly assigned to local variable *var*, then *var* is inferred to point to *heap*.

The analysis integrates two features that merit discussion, since they represent standard variability axes in the literature [Ryder, 2003]: *field sensitivity* and *on-the-fly call-graph construction*.

- Field sensitivity refers to the ability of the analysis to distinguish different fields of the same abstract object, instead of lumping all fields together. This is reflected in our rules for handling Store and Load instructions: a Store input fact, together with prior VarPointsTo inferences, leads to computing a FldPointsTo relation fact for the given heap object and field. The FldPointsTo facts are later used in the handling of Load instructions, as the respective rule shows.

[3] Reachable is somewhat of a special case, since we assume it is also used as an input relation: it needs to initially hold methods that are always reachable, such as the programs's `main` method. We ignore this technicality in the model, rather than burden our rules with a separate input relation.

```
VarPointsTo(var, heap) ←
 Reachable(meth), Alloc(var, heap, meth).

VarPointsTo(to, heap) ←
 Move(to, from), VarPointsTo(from, heap).

FldPointsTo(baseH, fld, heap) ←
 Store(base, fld, from), VarPointsTo(from, heap),
 VarPointsTo(base, baseH).

VarPointsTo(to, heap) ←
 Load(to, base, fld), VarPointsTo(base, baseH),
 FldPointsTo(baseH, fld, heap).


Reachable(toMeth),
VarPointsTo(this, heap),
CallGraph(invo, toMeth) ←
 VCall(base, sig, invo, inMeth), Reachable(inMeth),
 VarPointsTo(base, heap),
 HeapType(heap, heapT), Lookup(heapT, sig, toMeth),
 ThisVar(toMeth, this).

InterProcAssign(to, from) ←
 CallGraph(invo, meth),
 FormalArg(meth, n, to), ActualArg(invo, n, from).

InterProcAssign(to, from) ←
 CallGraph(invo, meth),
 FormalReturn(meth, from), ActualReturn(invo, to).

VarPointsTo(to, heap) ←
 InterProcAssign(to, from),
 VarPointsTo(from, heap).
```

Figure 2.2: Datalog rules for an Andersen-style points-to analysis and call-graph construction.

The alternative would be a *field-insensitive* analysis, which ignores which object field is used in a STORE or LOAD. This handling may sacrifice precision in an effort to keep the size of inferred facts more manageable.

Yet another option is a *field-based* analysis [Heintze and Tardieu, 2001a], which distinguishes fields but only identifies FLDPOINTSTO facts by the heap object's type and not its full identity. That is, fields of different heap objects of the same type are merged, although the fields themselves are kept separate, unlike in a field-insensitive analysis.

Without extensive experimentation with real programs, it is hard to predict whether such tradeoffs will pay off. Performance often exhibits sharp discontinuities: the lack of precision may end up also hurting performance, since the manipulated points-to sets can be larger.

- On-the-fly call-graph construction refers to the property that a points-to analysis also infers simultaneously which methods are called at each call-site. The two questions of "what objects can a variable point to?" and "which function can get called at this call-site?" are closely inter-related in any higher-order language. The target of, e.g., a Java call `obj.fun()` is only possible to resolve once the type of `obj` is known. Generally, virtual calls in object-oriented languages and first-class functions in a functional language necessitate that call-graph computation have a model of points-to information. On-the-fly call-graph construction typically leads to a significantly more precise call-graph, which, in turn, enhances the precision of the points-to analysis. The two analyses are in a virtuous circle, with each helping the other achieve precision without wasting effort in spurious inferences.

  On-the-fly call-graph construction is reflected in the rule for VCALL of Figure 2.2. This is the most involved rule of our Andersen-style algorithm. It states that, if the program has an instruction making a virtual method call over local variable *base* (this is an input fact), and if the computation so far has es-

tablished that *base* can point to heap object *heap* in a reachable method, then the called method is looked up by-signature inside the type of *heap* and several further facts are inferred: that the looked up method is reachable, that it has an edge in the call-graph from the current invocation site, and that its *this* variable can point to *heap*.

Furthermore, the computed call-graph information is used to flow points-to information between actual and formal parameters of methods, as shown in the last three rules of Figure 2.2. For instance, if we have computed a call-graph edge between invocation site *invo* and method *meth*, then we infer an inter-procedural assignment to the i-th formal argument of *meth* from the i-th actual argument at *invo*, for every i. Such assignments are treated much like local assignments (MOVE instructions) in the last rule.

Although on-the-fly call-graph construction is typically advantageous, there are several alternatives in the literature—e.g., see Tip and Palsberg [2000] for a comparative presentation. A conservative *a priori* call-graph computation has the advantage of speed, as well as of easier handling of complex language features (e.g., reflective calls, discussed in Chapter 3). The techniques of *Rapid Type Analysis (RTA)* [Bacon and Sweeney, 1996] and *Class Hierarchy Analysis (CHA)* [Dean et al., 1995] are the most commonly used in this space. Both techniques employ only type information to determine an over-approximation of all methods potentially called at a site.

Rule evaluation. The Datalog rules of our Andersen-style analysis are pure (with no constructors) and monotonic. Recall that this ensures a declarative specification: the order of evaluation of rules or examination of clauses cannot affect the analysis outcome.

For our code example at the beginning of the chapter, the algorithm would infer an approximation of the fully-precise result discussed earlier:

`fun1::a1` → `new A1()`

`fun1::b1` → `new A1(), new A2()`

`fun2::a2` → `new A2()`
`fun2::b2` → `new A1()`, `new A2()`
`id::a` → `new A1()`, `new A2()`

The result is not fully precise because it computes extra, spurious facts: that `fun1::b1` can point to abstract object `new A2()` (and not just to `new A1()`) and that `fun2:b2` can point to abstract object `new A1()` (and not just to `new A2()`). Let us consider how these inferences are made. The rules for handling inter-procedural assignments effectively emulate an assignment from an actual method parameter to a formal one, and from a formal return variable (inside the called procedure) to an actual one (inside a caller). For method `id`, this means that any elements of the points-to set of `id::a` will become elements of the points-to sets of `fun1::b1` and `fun2::b2`, since the latter are actual return variables for calls to `id` and `id::a` is a formal return variable.

The analysis loses precision relative to an ideal result, but this is hardly a surprise. Any algorithm for an undecidable problem will sacrifice precision for computability. Notably, the algorithm is still *sound*, i.e., if a fact is true the algorithm will infer it. Generally, pointer analysis is a *may-analysis*: its inferences intend to *over-approximate* actual program behavior. In Chapter 5 we will also see an *under-approximate*, *must-analysis*.

2.2 Other Approaches

Our Andersen-style analysis model is an excellent presentation vehicle and base for exploration of different techniques. Nevertheless, it does not fully capture the considerable variability of the pointer analysis space. We next discuss the main techniques that the model does *not* capture well. The contrast serves to illustrate different approaches.

Steensgaard's analysis. The best-known non-Andersen family of points-to analysis algorithms is that based on the work of Steensgaard [1996]. Steensgaard-style pointer analysis is best termed *unification-based* and uses *equality constraints* as opposed to the subset constraints of the Andersen approach. Specifically, a Steensgaard-style analysis

treats every input program statement as an indication that some points-to sets should be unified, i.e., become one. For example, in an Andersen-style analysis, a MOVE instruction, "`p = q`", leads to the points-to set of variable `q` becoming a subset of the points-to set of `p`. In a Steensgaard-style analysis, the same MOVE statement leads to the two points-to sets being merged, i.e., considered equal.

This approach of progressively merging points-to sets extends to the analysis of all language features. In a straightforward Steensgaard-style analysis, no abstract object can belong in more than one points-to set: sharing an object causes the unification of two points-to sets. Consider our code example at the beginning of the chapter. A Steensgaard-style algorithm (without further precision enhancements) would infer that all points-to sets are equal, since `fun1::a1` and `fun2::a2` are assigned to `id::a` and the latter is assigned back to `fun1::b1` and `fun2::b2`. The result is quite imprecise:[4]

`fun1::a1` → `new A1(), new A2()`
`fun1::b1` → `new A1(), new A2()`
`fun2::a2` → `new A1(), new A2()`
`fun2::b2` → `new A1(), new A2()`
`id::a` → `new A1(), new A2()`

The semantics of a Steensgaard-style analysis can be easily expressed in our Datalog framework. For instance, whereas the Andersen-style rule for a MOVE instruction is:

VARPOINTSTO(*to, heap*) ←
  MOVE(*to, from*), VARPOINTSTO(*from, heap*).

the corresponding Steensgaard-style analysis would also contain a symmetric rule:

VARPOINTSTO(*from, heap*) ←
  MOVE(*to, from*), VARPOINTSTO(*to, heap*).

Expressing a Steensgaard-style points-to analysis in Datalog is rather misleading, however. For the analyses we present in our Datalog

[4]Indeed, it is often necessary to artificially limit the application of Steensgaard-style equality constraints, or all points-to sets would be equal. One such case is the constructor for a top-of-type-hierarchy class in an object-oriented language: conceptually, all abstract objects are used as the `this` variable in such a constructor call, so a naive treatment would have all points-to sets be merged.

models, implementation is straightforward: a standard evaluation of the Datalog program matches the asymptotic complexity (if not the practical efficiency) of state-of-the-art manual implementations. This is not the case for a Steensgaard-style analysis, which emphasizes high performance as its hallmark property. Since every abstract object can appear in at most one points-to set, the analysis can execute in practically-linear time (relative to the number of input instructions) with the use of union-find trees. Operations on union-find trees are of amortized near-constant complexity, and every program statement induces a constant number of membership checks and set union operations. A straightforward Datalog execution of the corresponding rules cannot match such performance, as it will merely copy elements from one points-to set to another, instead of efficiently unifying their representations.

Overall, Steensgaard-style analyses have been quite popular, especially in procedural languages such as C, due to their simplicity and unparalleled speed. However, they have become progressively less used in recent programming languages and modern settings, where the speed of an Andersen-style analysis is usually quite sufficient.

Alias analysis, other heap abstractions. Although we, and the majority of the literature, treat "pointer analysis", "points-to analysis", and "alias analysis" as virtual synonyms, there is occasionally reason to focus on their subtle differences. Hind [2001] distinguishes between alias and points-to analysis, depending on whether an algorithm intends to compute alias pairs or points-to sets, respectively. (Recall that points-to analysis attempts to answer questions of the form "what is the set of abstract objects that this variable/expression may point to?" whereas alias analysis answers the question "can these two variables/expressions denote the same object?") "Pointer analysis" is then the union of "points-to analysis" and "alias analysis". In this view, alias analysis may not need explicit representations of heap objects (e.g., abstract objects represented as allocation sites) but instead can directly compute aliased pairs of variables. Some algorithms, especially on the high-speed, lower-precision end of the spectrum—e.g., [Zheng and Rug-

ina, 2008], are explicitly designed to compute such alias relationships very efficiently.

More generally, not just variables but entire expressions, or syntactic abstractions over them, can be used as the root elements of an analysis. This obviates the need to represent allocation sites or objects explicitly. The *access path* abstraction approach generalizes alias relations by expressing all knowledge about the heap in terms of relationships between pointer expressions. An access path is a variable qualified by a sequence of field names. Access paths can be statically summarized syntactically—e.g., using regular expressions, such as `p.f.*` to mean any path starting with `p.f`—in order to capture unbounded runtime field chains. An analysis can use access paths internally or in expressing its computed output. Typically, analyses maintain equivalence classes of access paths—e.g., storing that `p.f` and `q.g.h` can be aliases—although other predicates on access paths may also be profitable to compute. Kanvar and Khedker [2014] discuss exhaustively a variety of models for the heap.

3

Analysis of Realistic Languages

Our model of a pointer analysis forms a firm base on which more precise and more complete algorithms can be built. In this chapter we discuss completeness: how can different language features be modeled in the course of a pointer analysis? We next see the additions to a core analysis model that are required. We discuss a variety of languageq features, each with its own handling but all painting from a common palette, as simple extensions of our base points-to analysis. There are interesting subtleties, especially concerning features (e.g., exceptions, reflection) that essentially represent separate analyses, yet are best handled on-the-fly, in mutual recursion with the core analysis.

3.1 Arrays and Other Language Features

Our intermediate language of Section 2.1 ignores several features of a realistic programming language. Some can be treated entirely analogously to features shown. For instance, static method calls can be represented in the input intermediate representation via a separate re-

lation (SCall(*sig: S, invo: I, inMeth: M*)) and handled using a subset of the logic for virtual calls in Figure 2.2.[1]

A particularly central and interesting language feature for pointer analysis purposes is arrays. One of the simplest and most common approaches to array modeling in a pointer analysis is *array-insensitivity*. This signifies that the analysis does not distinguish between loads and stores to different array locations. The result is that precision is lost, yet the analysis reasoning remains entirely at the level of variables and heap objects: no reasoning on integer expressions is required.

We can add handling of arrays (for a Java-like language) to our base pointer analysis model quite straightforwardly. Figure 3.1 lists new input and computed relations, as additions to those in Figure 2.1. ArrayLoad and ArrayStore add to our intemediate language the usual array access instructions. ComponentType represents the type system information that reflects the element type of an array. (For instance, the component type of `A[][]` is `A[]` and *its* component type is `A`.) A new computed relation ArrayContentsPointTo computes the points-to set of an array object, by analogy to the earlier FldPointsTo predicate for plain heap objects.

ArrayLoad(*to : V, base : V*) # *to = base[...]*
ArrayStore(*base : V, from : V*) # *base[...] = from*

ComponentType(*type : T, compT : T*)

ArrayContentsPointTo(*baseH : H, heap : H*)

Figure 3.1: Extra input and computed relations (added to Figure 2.1) for array instructions. Note that the index expression is omitted from the ArrayLoad and ArrayStore predicates, since it is completely redundant with our array-insensitive approach. More sophisticated analyses would need such information.

Figure 3.2 shows the analysis enhancements for array instructions, with rules to be viewed as additions to Figure 2.2. Input instructions

[1] In more precise, context-sensitive, analyses—see Chapter 4—Kastrinis and Smaragdakis [2013b] argue that it is profitable to distinguish between static and virtual calls.

ArrayLoad and ArrayStore are handled by separate rules, with the computed ArrayContentsPointTo relation to link them. As can be seen, the logic for array handling is mutually recursive with the core points-to analysis logic: relation ArrayContentsPointTo is established based on VarPointsTo facts, and vice versa.

```
ArrayContentsPointTo(baseH, heap) ←
  ArrayStore(base, from),
  VarPointsTo(base, baseH), VarPointsTo(from, heap),
  HeapType(heap, hType), HeapType(baseH, baseHType),
  ComponentType(baseHtype, componentType),
  Subtype(hType, componentType).

VarPointsTo(to, heap) ←
  ArrayLoad(to, base),
  VarPointsTo(base, baseH), ArrayContentsPointTo(baseH, heap).
```

Figure 3.2: Datalog rules (additions to Figure 2.2) for array modeling.

Notably, type information is used for filtering which abstract objects flow into an array and out of it. In the first rule, the component type of the concrete type of an abstract array object is retrieved and needs to be compatible with (i.e., a supertype of) the type of the object stored in the array. Type checks such as this are often important for precision. Due to the inherent imprecision of an array-insensitive approach, a heavy burden is placed on the analysis to recover precision whenever possible in array manipulation.

3.2 Exception Analysis

Analyzing the exception flow of a program is often treated as a separate, high-level analysis—as a client of a pointer analysis, rather than as an integral part of it [Fu and Ryder, 2007]. Exception flow information is of direct value for human inspection of the program, thus reinforcing the view of exception analysis as a high-level client. Nevertheless, experimental results for Java programs have shown that exception analysis and pointer analysis often enjoy the same symbiotic relationship as

pointer analysis and (on-the-fly) call-graph construction: the precision of either analysis is significantly harmed when they are performed separately, with conservative assumptions about the other. Bravenboer and Smaragdakis [2009b] show that some flavors of pointer analyses (esp. *object-sensitive* analyses, which we discuss in Section 4.3) can be many times faster and twice as precise with an on-the-fly exception analysis, performed in mutual recursion with the pointer analysis logic.

Figure 3.3 shows additions to our baseline analysis inputs and computed relations for exception analysis. The input relations encode Java-like exception *throwing* and *catching* instructions in an input program. The THROW(*i,e*) relation captures throwing at instruction i an expression object that is referenced by local variable e. The CATCH(*t,i,a*) relation connects an instruction i, which can (directly or via a call) throw an exception of dynamic type t, with the local variable, a, that will be assigned the exception object at the appropriate catch-site. Although CATCH does not directly map to individual intermediate language instructions, one can compute it easily from such low-level input. (The existence of a `catch` instruction is a definite fact, deduced from direct syntactic inspection of the input program, and not an analysis inference.) Introducing CATCH as a slightly abstracted input relation allows the modeling of exception handlers at different degrees of precision—e.g., a definition of CATCH may or may not consider exception handlers in-order, may or may not consider only the most-specific handler based on the declared types, etc. The output relation we want to compute is THROWPOINTSTO, which captures what exception objects a method may throw at its callers.

THROW(*instr : I, e : V*)	*# throw(e)*
CATCH(*heapT : T, instr : I, arg : V*)	*# catch(arg)*
THROWPOINTSTO(*meth : M, heap : H*)	

Figure 3.3: Extra input and computed relations (added to Figure 2.1) for exception analysis.

```
ThrowPointsTo(meth, heap) ←
  InMethod(instr, meth), Throw(instr, e),
  VarPointsTo(e, heap), HeapType(heap, heapT),
  !Catch(heapT, instr, _).

ThrowPointsTo(meth, heap) ←
  InMethod(invo, meth), CallGraph(invo, toMeth),
  ThrowPointsTo(toMeth, heap), HeapType(heap, heapT),
  !Catch(heapT, invo, _),

VarPointsTo(arg, heap) ←
  Throw(instr, e), VarPointsTo(e, heap),
  HeapType(heap, heapT), Catch(heapT, instr, arg).

VarPointsTo(arg, heap) ←
  CallGraph(invo, toMeth),
  ThrowPointsTo(toMeth, heap),
  HeapType(heap, heapT), Catch(heapT, invo, arg).
```

Figure 3.4: Datalog rules (additions to Figure 2.2) for exception analysis.

Figure 3.4 shows the exception computation, in mutual recursion with the points-to analysis. Two syntactic constructs we have not seen before are "_", meaning "any value",[2] and "!", signifying negation.[3]

ThrowPointsTo is our new output relation, which also leads to more VarPointsTo inferences: rules for one relation appeal to the other. The first rule captures the case of a thrown exception when no

[2]Semantically, "_" is just a regular logical variable, albeit one that does not need to have a human-readable name. Different instances of "_" are distinct.

[3]This use of negation is merely a syntactic convenience, since it only applies to input relations (Catch), which could also have been negated in advance. As expected, based on the semantics of "_", when an "_" variable is in a negated predicate, the meaning is "there does not exist a value such that...".

Generally, our rules respect the restriction of *stratified negation*: negation does not appear in a recursive cycle. This restriction is a standard assumption for the polynomial-time execution guarantees of the Datalog language discussed in Chapter 1.

matching `catch` instruction exists in the same method. In this case, the method itself can throw the exception to its callers. The second rule handles the propagation of an exception thrown by a transitively called method, in the case that no matching handler exists in the caller. The third and fourth rules model the complementary case of exceptions thrown locally or by transitively called methods, but caught. The result is a VarPointsTo inference for the variable to which the exception object is assigned in the `catch` clause.

In practice, exception objects may be represented more coarsely than other program objects, leading to scalability gains [Kastrinis and Smaragdakis, 2013a]. This approach is straightforward to implement in the above declarative framework. No changes to the exception-handling logic are needed—instead, we can merely adjust how `new` instructions imply VarPointsTo facts, by distinguishing two cases (coarse- vs. fine-grained representation) in the first rule of Figure 2.2, based on the type of the allocated object.

3.3 Reflection Analysis

One of the most significant practical issues facing pointer analysis is the treatment of dynamic language features: reflection, dynamic loading, dynamic changes to the member lookup process. Such features are prevalent in dynamic languages (e.g., JavaScript, Python, PHP) yet also play an ever-increasing role in statically-typed, two-phase-compiled languages (Java, C#, and more). Few practical pointer analysis approaches currently attempt to statically model dynamic language features: the typical design choice is to pretend that the dynamic features are absent, thus sacrificing the soundness (i.e., property of over-approximation) of a may-analysis [Livshits et al., 2015]. Nevertheless, the handling of dynamic features is a promising research frontier that increasingly receives attention.

Among dynamic features, the best-studied is *reflection* [Livshits, 2006, Li et al., 2014]. Reflection, as in the Java language, refers to the ability to dynamically access members and type information of an object, often based on string representations of the member's or type's

name. We next reformulate the reflection analysis approach introduced by Livshits et al. [2005] for Java. This treatment of reflection is interesting because it is elegant, practically feasible, and highly representative of disciplined handling of dynamic features in other languages (e.g., JavaScript [Madsen et al., 2013]).

A key to the approach is that, much like with earlier language features, reflection analysis relies on points-to information, and vice versa. A typical pattern of reflection usage in Java is with code such as:

```
String className = ... ;      // possibly a constant string
Class c = Class.forName(className);
Object o = c.newInstance();
String methodName = ... ;     // possibly a constant string
Method m = c.getMethod(methodName, ...);
m.invoke(o, ...);
```

In the above, a string is used to dynamically look up a class. The result of the lookup is a *class object*, uniquely identifying the class at run-time. The class is then instantiated, yielding an object. A method with a name given by a string is looked up in the class (producing a *method object*) and invoked on the newly-generated object.

The insight behind the static reflection analysis is that the program text contains string constants[4] that can often determine the outcome of reflection operations. Thus, the challenge becomes to track the flow of string constants through the program. In our above example, all of the statements can occur in far-away program locations, across different methods, invoked through virtual calls from multiple sites, etc. Thus, a whole-program analysis with an understanding of heap objects is required to track reflection with any amount of precision. This suggest the idea that reflection analysis can leverage points-to analysis—it is a client for points-to information. At the same time, points-to analysis needs the results of reflection analysis—e.g., to determine which method gets invoked in the last line of the above example, or what

[4]More sophisticated analyses (e.g., [Li et al., 2014]) may also perform substring matching, may track the flow of strings through library methods that produce new strings, etc. For simplicity, we only model the essence of the analysis, for the simple case of full string constants.

objects each of the example's local variables point to. Thus, reflection analysis and points-to analysis become mutually recursive, or effectively a single analysis.

We consider the core of the reflection analysis algorithm, which handles the most common reflection features for the Java language, illustrated in our above example: creating a class object given a name string (library method `Class.forName`), creating a new object given a class object (library method `Class.newInstance`), retrieving a method object given a class object and a signature (library method `Class.getMethod`), and reflectively calling a virtual method on an object (library method `Method.invoke`).[5]

Reflection analysis adds a few extra (auxiliary) input relations to the base pointer analysis, as shown in Figure 3.5.

ConstantForClass(*h: H, t: T*)
ConstantForMethod(*h: H, sig: S*)
ReifiedClass(*t: T, h: H*)
ReifiedMethod(*sig: S, h: H*)
ReifiedHeapAllocation(*i: I, t: T, h: H*)

Figure 3.5: Extra input relations (added to Figure 2.1) for reflection analysis.

The ConstantForClass(*h: H, t: T*) relation encodes that class/type t has a name represented by the constant string object h in the program text, and similarly for relation ConstantForMethod(*h: H, sig: S*) and method signature *sig*. Only a small subset of the classes or methods in the program will have names represented as constant strings in the program text.

The reflection analysis also needs to invent new abstract objects, which do not correspond to Alloc instructions in the program text. We assume that these are supplied as analysis inputs, via straightforward pre-processing of the program text. Relation ReifiedClass(*t: T, h: H*)

[5]This treatment ignores several other API calls, which are handled similarly. These include, for instance, the handling of fields and constructors, other kinds of method invocations (static, special), reflective access to arrays, other ways to get class objects, and more.

links an abstract class object, h, with the class type, t, it represents. (Thus, the static analysis uses a unique abstract object per dynamic class object.) Similarly, relation ReifiedMethod(*sig: S, h: H*) links an abstract method object, h, with method signature *sig*. (The signature includes the declaring type.) Finally, ReifiedHeapAllocation(*i: I, t: T, h: H*) returns an abstract object, h, to represent all dynamic objects of type t that are allocated with a `newInstance` call at invocation site i.

Figure 3.6 shows the reflection analysis additions to our base pointer analysis algorithm. To simplify the presentation we consider standard strings (e.g., `"Class.forName"`) to stand for the signature of the respective method. Note also the use of a static call (relation SCall, as discussed in Section 3.1) in the first rule.

The first rule says that if the first argument (0-th parameter of the static call) of a `forName` call points to an object that is a string constant matching a class name, then the local variable, r, receiving the return value of the `forName` call will point to the abstract class object, h, for the class.

The second rule handles the case of a `newInstance` call. If the receiver object, h_c, of the call is a class object for class t, and the `newInstance` call is assigned to variable r, then r can point to the special allocation site, h, that designates objects of type t allocated at the `newInstance` call site.

The third rule gives semantics to `getMethod` calls. It states that if such a call is made with receiver b (for "base") and first argument p (the string encoding the desired method's signature), and if the analysis has already determined the objects that b and p may point to, then, assuming p points to a string constant encoding a signature, s, that exists inside the type that b points to, the variable r holding the result of the `getMethod` call points to the reflective object, h_m, for this method signature.

Finally, the fourth rule uses reflection information to infer more call-graph edges. A new edge can be inferred from the invocation site, i, of a reflective `invoke` call to a method m, if the receiver, b, of the `invoke` call points to a reflective object encoding a method signature,

VarPointsTo(r, h) ←
SCall(*"Class.forName"*, i, _), ActualReturn(i, r)
ActualArg(i, *0*, p), VarPointsTo(p, c),
ConstantForClass(c, t), ReifiedClass(t, h).

VarPointsTo(r, h) ←
VCall(v, *"Class.newInstance"*, i, _),
VarPointsTo(v, h_c), ReifiedClass(t, h_c),
ActualReturn(i, r), ReifiedHeapAllocation(i, t, h).

VarPointsTo(r, h_m) ←
VCall(b, *"Class.getMethod"*, i, _), ActualReturn(i, r),
VarPointsTo(b, h_c), ReifiedClass(t, h_c),
ActualArg(i, *1*, p),
VarPointsTo(p, c), ConstantForMethod(c, s),
LookUp(t, s, _), ReifiedMethod(s, h_m).

CallGraphEdge(i, m) ←
VCall(b, *"Method.invoke"*, i, _),
VarPointsTo(b, h_m), ReifiedMethod(s, h_m),
ActualArg(i, *1*, p), VarPointsTo(p, h),
HeapType(h, t), LookUp(t, s, m).

Figure 3.6: Datalog rules (additions to Figure 2.2) for reflection analysis.

and the first argument, p, of the `invoke` call (the intended receiver of the reflective invocation) points to an object, h, of a class in which the lookup of the signature produces the method m.

Although the above rules are occasionally intricate, they are also concise and elegantly enhance existing pointer analysis algorithms. The rules offer a minimal but faithful model of the core of a practical reflection analysis, performed as part of a pointer analysis.

Discussion. As the preceding sections have shown, much of the complexity of pointer analysis is due to the handling of modern language

features. These features are not orthogonal: static analysis is a domain best described as groups of mutually recursive definitions, each handling a certain language feature. The handling of each feature—e.g., exceptions or reflection—produces more facts for the rest of the analysis, while, vice versa, facts produced by the rest of the analysis trigger more inferences for the exception or reflection logic. This consideration is reflected in practice. The effectiveness of a pointer analysis algorithm, in terms of both precision and performance, may be dramatically affected by the modeling of extra language features. An algorithm that appears superior on a core language model may well be inferior once more of the language features are modeled. The complexity of such inter-related analyses is also an impetus for declarative modeling, as in our approach. A declarative specification of the analysis allows for modular and compact models: none of the specifications of this Chapter required anything but modular additions to the original pointer analysis framework of Figures 2.1 and 2.2.

4

Context Sensitivity

Pointer analysis attempts to offer efficiently computable yet reasonably precise models of the program heap. The inter-related and whole-program nature of the analyses for different language features practically means that intractability lurks behind even the simplest efforts to maintain high precision. Although the algorithms we examine in this tutorial are polynomial (as evidenced by their specification in Datalog) they can quickly become unscalable for realistic programs, analyzed against modern, large standard libraries.

For higher-order (object-oriented and functional) languages, the approach of *context sensitivity* has arisen as the primary way to enhance analysis precision without sacrificing scalability.[1] A context-sensitive analysis qualifies variables and abstract objects with context information: the analysis collapses information (e.g., "what objects this local variable can point to") over executions that map to the same context value, while separating executions that map to different contexts. This offers a way to control the precision of the analysis and even select the

[1] Context sensitivity is of value even to first-order function/procedure calls, but in such a language (e.g., C) researchers often derive greater value from *flow sensitivity*—an approach we discuss in Chapter 5.

precision aspects more pertinent to the analysis at hand. Context sensitivity comes in many flavors, such as *call-site sensitivity* [Sharir and Pnueli, 1981, Shivers, 1991], *object sensitivity* [Milanova et al., 2002, 2005], and *type sensitivity* [Smaragdakis et al., 2011], depending on what program element is used as context information.

We begin with a general model for context sensitivity. We shall instantiate the model to discuss various flavors.

4.1 Context-Sensitive Analysis Model

To motivate context-sensitive analysis, consider our earlier example from Chapter 2, partly reproduced below:

```
void fun1() {
  Object a1 = new A1();
  Object b1 = id(a1);
}
... // and similarly for a fun2
Object id(Object a) { return a; }
```

The Andersen-style analysis of Section 2.1 is context-insensitive. We would like to modify it to become context-sensitive, thus achieving more precision. As we saw, the context-insensitive analysis result differs from the ideal, fully precise one—e.g., `fun1::b1` is computed to point to both abstract objects `new A1()` and `new A2()`, whereas in reality it can only point to the former.

In order to achieve more precision, we can qualify the analysis inferences with context information. There are two kinds of context: *calling context* (or often just *context*), which is used to qualify local variables, and *heap context*, which is used to qualify heap abstractions. That is, if we have two calling contexts *c1* and *c2* for method `id`, the analysis will compute separately the points-to information of local variable `a` under context *c1* and under context *c2*. Similarly, if we have two heap contexts *hc1* and *hc2* at the point of the allocation instruction `new A1()`, we will distinguish the allocated object for *hc1* from that for *hc2*. This distinction will follow the abstract object wherever it flows in the analysis.

In later sections, we shall make the above illustration more concrete, for specific kinds of context information.

To describe context-sensitive analyses in our declarative model, we merely need to augment our computed relations with context information. There are two kinds of context sets in our domain: a set of calling contexts (or just "contexts"), C, and a set of heap contexts, HC.

Figure 4.1 shows the updated input domain and schema of computed relations for a context-sensitive analysis. Most of the figure reproduces Figure 2.1, for ease of reference. The only additions in the first three parts of Figure 4.1 are those of sets C and HC, as well as that of adding context parameters in the signatures of computed relations. The fourth part lists two new special relations, **Record** and **Merge** serving as *constructors* of contexts. A constructor returns a context object for each combination of input values.

Figure 4.2 updates Figure 2.2 for a context-sensitive analysis. As can be seen, the rules are isomoprhic to the context-insensitive rules, with only straightforward modifications.

The main difference in the addition of context sensitivity to the analysis logic is the use of constructors **Record** and **Merge**. As far as the rest of the analysis is concerned, **Record** and **Merge** are black boxes. Their use makes our analysis parametric. Different implementations of **Record** and **Merge** will yield analyses of different context sensitivity flavors.

Record is employed in the handling of ALLOC instructions (i.e., `new` statements in Java). Every time an allocation site is encountered, the abstract object for the allocation site is qualified with the heap context that **Record** returns. This provides more fine-grained differentiation than merely using allocation sites as abstractions for objects. Note that the interface of **Record** (i.e., its two arguments, *heap* and *ctx*) is general enough to capture all information available to the Datalog rule for ALLOC. (For a given heap object, *heap*, both the method *meth* that allocates it, and the variable *var*, to which it is assigned, can be uniquely determined by the identity of this heap object, i.e., there exists a functional dependency from *heap* to *var* and *meth*. Thus, the latter can be omitted from the signature of **Record**.) In other words, **Record**

V is a set of program variables
H is a set of heap abstractions (i.e., allocation sites)
M is a set of method identifiers
S is a set of method signatures (including name, type signature)
F is a set of fields
I is a set of instructions (mainly used for invocation sites)
T is a set of class types
$\mathbb{N}$ is the set of natural numbers
C is a set of (calling) contexts
HC is a set of heap contexts

ALLOC(*var : V, heap : H, inMeth : M*)	*# var = new ...*
MOVE(*to : V, from : V*)	*# to = from*
LOAD(*to : V, base : V, fld : F*)	*# to = base.fld*
STORE(*base : V, fld : F, from : V*)	*# base.fld = from*
VCALL(*base : V, sig : S, invo : I, inMeth : M*)	*# base.sig(..)*

FORMALARG(*meth : M, n : $\mathbb{N}$, arg : V*)
ACTUALARG(*invo : I, n : $\mathbb{N}$, arg : V*)
FORMALRETURN(*meth : M, ret : V*)
ACTUALRETURN(*invo : I, var : V*)
THISVAR(*meth : M, this : V*)
HEAPTYPE(*heap : H, type : T*)
LOOKUP(*type : T, sig : S, meth : M*)
VARTYPE(*var: V, type: T*),
INMETHOD(*instr : I, meth : M*)
SUBTYPE(*type : T, superT : T*)

VARPOINTSTO(*var : V, ctx : C, heap : H, hctx : HC*)
CALLGRAPH(*invo : I, callerCtx : C, meth : M, calleeCtx : C*)
FLDPOINTSTO(*baseH : H, baseHCtx : HC, fld : F, heap : H, hctx : HC*)
INTERPROCASSIGN(*to : V, toCtx : C, from : V, fromCtx : C*)
REACHABLE(*meth : M, ctx : C*)

Record (*heap : H, ctx : C*) = *newHCtx : HC*
Merge (*heap : H, hctx : HC, invo : I, ctx : C*) = *newCtx : C*

Figure 4.1: Our domain, input relations, computed relations, and constructors of contexts. The input relations are of two kinds: relations encoding program instructions (the form of the instruction is shown in a comment), and relations encoding type system and other environment information.

Record (*heap, ctx*) = *hctx*,
VarPointsTo(*var, ctx, heap, hctx*) ←
 Reachable(*meth, ctx*), Alloc(*var, heap, meth*).

VarPointsTo(*to, ctx, heap, hctx*) ←
 Move(*to, from*), VarPointsTo(*from, ctx, heap, hctx*).

FldPointsTo(*baseH, baseHCtx, fld, heap, hctx*) ←
 Store(*base, fld, from*), VarPointsTo(*from, ctx, heap, hctx*),
 VarPointsTo(*base, ctx, baseH, baseHCtx*).

VarPointsTo(*to, ctx, heap, hctx*) ←
 Load(*to, base, fld*), VarPointsTo(*base, ctx, baseH, baseHCtx*),
 FldPointsTo(*baseH, baseHCtx, fld, heap, hctx*).

Merge (*heap, hctx, invo, callerCtx*) = *calleeCtx*,
Reachable(*toMeth, calleeCtx*),
VarPointsTo(*this, calleeCtx, heap, hctx*),
CallGraph(*invo, callerCtx, toMeth, calleeCtx*) ←
 VCall(*base, sig, invo, inMeth*), Reachable(*inMeth, callerCtx*),
 VarPointsTo(*base, callerCtx, heap, hctx*),
 HeapType(*heap, heapT*), LookUp(*heapT, sig, toMeth*),
 ThisVar(*toMeth, this*).

InterProcAssign(*to, calleeCtx, from, callerCtx*) ←
 CallGraph(*invo, callerCtx, meth, calleeCtx*),
 FormalArg(*meth, n, to*), ActualArg(*invo, n, from*).

InterProcAssign(*to, callerCtx, from, calleeCtx*) ←
 CallGraph(*invo, callerCtx, meth, calleeCtx*),
 FormalReturn(*meth, from*), ActualReturn(*invo, to*).

VarPointsTo(*to, toCtx, heap, hctx*) ←
 InterProcAssign(*to, toCtx, from, fromCtx*),
 VarPointsTo(*from, fromCtx, heap, hctx*).

Figure 4.2: Datalog rules for context-sensitive Andersen-style points-to analysis and call-graph construction.

has all possible inputs available to it, and its specific definition can choose what it wants to use as context. We will see in the next sections several instances of such choices.

Merge is similar to **Record**. It takes all available information at the *call site* of a method and combines it to create a new calling context (or just "context"). In this way, variables that pertain to different invocations of the same method are distinguished, as much as the context granularity allows.

Importantly, the addition of constructors to Datalog makes the language Turing-complete, i.e., invalidates the polynomial-time guarantee for expressible programs. Therefore, care must be taken to ensure that the analysis terminates. Both our constructors **Record** and **Merge** are recursive: they take as input entities of the same type as the one they return. Therefore, we cannot preclude their use in non-terminating computations. To restore our target property of polynomial execution, in the next sections we limit our attention to definitions of **Record** and **Merge** that create contexts in domains isomorphic to finite sets, bounded polynomially by the size of the input. (For instance, we may define the contexts of a given analysis to be of the form "pairs of allocation sites", or "triples of allocation site $\times$ call site $\times$ type", etc.)

4.2 Call-Site Sensitivity

Call-site sensitivity [Sharir and Pnueli, 1981, Shivers, 1991] is the oldest and best-known flavor of context sensitivity. It consists of using call sites as context. Namely, whenever a method gets called, the context under which the called method gets analyzed by a k-call-site-sensitive analysis is a sequence of call sites: the current call site of the method, the call site of the caller method, the call site of the caller method's caller, etc., up to a pre-defined depth, k.

To illustrate, let us consider again our earlier example from Chapter 2, reproduced in full below for ease of reference.

```
void fun1() {
  Object a1 = new A1();
  Object b1 = id(a1);
}
```

```
void fun2() {
  Object a2 = new A2();
  Object b2 = id(a2);
}
Object id(Object a) { return a; }
```

As we saw in Chapter 2, a context-insensitive Andersen-style analysis produces an over-approximation of the fully precise result. The return values of the two calls to `id` are conflated, yielding spurious assignments to local variables `b1` and `b2`:

`fun1::a1` → `new A1()`
`fun1::b1` → `new A1(), new A2()`
`fun2::a2` → `new A2()`
`fun2::b2` → `new A1(), new A2()`
`id::a` → `new A1(), new A2()`

A call-site sensitive analysis addresses this imprecision. A 1-call-site-sensitive analysis, for instance, qualifies local variable `id::a` by the call site of method `id`. This yields analysis results of the form:

`fun1::a1` → `new A1()`
`fun1::b1` → `new A1()`
`fun2::a2` → `new A2()`
`fun2::b2` → `new A2()`
`[1]id::a` → `new A1()`
`[2]id::a` → `new A2()`

Note how local variable `id::a` is now analyzed in two contexts. A projection of the context-sensitive results to context-insensitive ones would merge the two points-to sets of qualified variables `[1]id::a` and `[2]id::a`, but internally the analysis has kept the distinction, which, in turn, has resulted in computing the fully-precise result for `fun1::b1` and `fun2::b2`.

Call-site-sensitive analyses can be expressed in our Datalog model with appropriate instantiations of constructors **Record** and **Merge**. The simplest kind of analysis is 1-call-site sensitive, without any context for heap objects. Such an analysis has $C = I$ (i.e., contexts are instructions, and specifically call instructions) and $HC = \{\star\}$ (where $\star$ is just a unique element, different from all others in our domains). The analysis is then defined simply by setting:

Record (*heap, ctx*) = $\star$
Merge (*heap, hctx, invo, ctx*) = *invo*

That is, the analysis keeps no context when an object is created (**Record**) while it keeps the invocation site as context in calls (**Merge**).

Similarly, we can define a 1-call-site-sensitive analysis with a *context-sensitive heap* (an approach also known as *heap cloning*). In this case, $HC = C = I$, and our constructors become:

Record (*heap, ctx*) = *ctx*
Merge (*heap, hctx, invo, ctx*) = *invo*

That is, the analysis uses the current method's context as a heap context for objects allocated inside the method. As before, the context for a method call is the call site.

Deeper contexts are equally easy to support, by deconstructing existing contexts and constructing more complex ones. For instance, a 2-call-site-sensitive analysis with a 1-call-site-sensitive heap would have $C = I \times I$, $HC = I$ and constructors:

Record (*heap, ctx*) = *first(ctx)*
Merge (*heap, hctx, invo, ctx*) = *pair(invo, first(ctx))*

(We assume standard functions such as *pair* and *first* to construct and deconstruct sequences.)

In all, call-site sensitivity has been a well-established approach in the literature and in analysis tools, for over two decades. As also seen in our example, however, its effectiveness relies on syntactic patterns in the program. For instance, if an extra level of indirection—e.g., a function `id2` that merely wraps `id` and is called instead of it—is added to the example, a 1-call-site sensitive analysis will fail to maintain precision. In experimental studies on object-oriented languages (e.g., [Lhoták and Hendren, 2008]), a different flavor of context sensitivity has been found to yield higher precision, and often better performance.

4.3 Object Sensitivity

Since its introduction [Milanova et al., 2002], object sensitivity has emerged as the dominant flavor of context sensitivity for object-oriented

languages. Object sensitivity uses object abstractions, i.e., *allocation sites*, as contexts. Specifically, the analysis qualifies a method's local variables with the allocation site of the *receiver object* of the method call. This kind of context information is non-local: it cannot be gathered by simple inspection of the call site, since it depends on what the analysis itself has computed to be the receiver object.

For our earlier example program fragment (Chapter 2 and Section 4.2) it is not clear whether object sensitivity might gain precision over a context-insensitive analysis: the implicit receiver (`this`) of the two calls to `id` may be the same or not, depending on the receivers of the calls to `fun1` and `fun2`. More of the program text is required in order to determine the result of the analysis. For illustration, we update the program, below, so that the receiver objects of the two calls are guaranteed distinct. Furthermore, we add an extra level of calling, via a method `id2`, which wraps `id`—a complication that would cause a loss of precision for a call-site-sensitive analysis.

```
class S {
  Object id(Object a) { return a; }
  Object id2(Object a) { return id(); }
}
class C extends S {
  void fun1() {
    Object a1 = new A1();
    Object b1 = id2(a1);
  }
}
class D extends S {
  void fun2() {
    Object a2 = new A2();
    Object b2 = id2(a2);
  }
}
```

An object-sensitive analysis maintains full precision for the above program fragment, even for a context consisting of a single object abstraction. The receiver objects of the two calls to `id2` (implicitly `this`) are distinct abstract objects (guaranteed in the example, since they are of different dynamic types, `C` and `D`). Therefore, the call to `id` in-

side `id2` is analyzed (at least) twice—once for each context/abstract receiver object. The result is that the abstract objects that flow to `b1` and `b2` are kept separate.

Much of the power of object sensitivity is due to this tolerance to extra levels of calling indirection, as long as the receiver object stays the same. In contrast, a call-site-sensitive analysis with context depth 1 would not have maintained full precision for the above example, exactly because of the addition of `id2` as an extra intermediate call. A call-site-sensitive analysis would require two levels of context to regain full precision. Still, this only offers limited protection: an extra level of calling indirection (e.g., an `id3` that wraps `id2`) would have necessitated a third level of call-site context for full precision, and so on.

Object-sensitive analyses can again be expressed quite simply in our model, with instantiations of constructors **Record** and **Merge**. The simplest kind is a 1-object-sensitive analysis without context for heap objects. This analysis sets $C = H$ and $HC = \{\star\}$, with constructors:

Record (*heap, ctx*) = $\star$
Merge (*heap, hctx, invo, ctx*) = *heap*

That is, the analysis stores no context for allocated objects. For method calls (refer to Figure 4.2), the context is the allocation site of the receiver object.

A more ambitious 2-object-sensitive analysis with a 1-context-sensitive heap has a heap context of one allocation site ($HC = H$) and a calling context of two allocation sites ($C = H \times H$). The respective constructors are:

Record (*heap, ctx*) = *first(ctx)*
Merge (*heap, hctx, invo, ctx*) = *pair(heap, hctx)*

Or: the context of a method call (determined by **Merge**) is a 2-element list consisting of the receiver object and its (heap) context. The heap context of an object (fixed at allocation, via **Record**) is the first context element of the allocating method, i.e., the receiver object on which it was invoked. Therefore, the context of a method call is the receiver

object together with the "parent" receiver object (the receiver object of the method that allocated the receiver object of the call).

Analyses with deeper context are defined similarly.

4.4 Discussion

In practical terms, context sensitivity can be quite effective [Lhoták and Hendren, 2008]. It yields a way to enhance precision significantly, often with little to no sacrifice of scalability.[2] Additionally, it is seamless to integrate with other analysis ideas. Although our formulation of context sensitivity has changed the signature of computed relations, the change is uniform: relations acquire context and heap context entries, as applicable. All analysis add-ons from Chapter 3 apply transparently to a context-sensitive analysis expressed in our formulation, with rather minimal and straightforward updates.

There are several more aspects of context sensitivity that are of interest. These include further algorithms, complexity results, and more.

Other flavors of context sensitivity. In addition to call-site sensitivity and object sensitivity, other flavors have been explored in the literature, and even seem to offer significant advantages.

Type sensitivity [Smaragdakis et al., 2011] is an approach directly isomorphic to object sensitivity, yet with types used in every place allocation sites would normally appear. These types represent the class containing the respective allocation site. Thus, allocation sites in methods declared in the same class (though not inherited methods) are merged. The goal of type sensitivity is to coarsen the context of an object-sensitive analysis, in order to achieve scalability (by avoiding replication of information between contexts) without much precision loss.

[2] It is often hard to tune the application of context-sensitivity—i.e., to pick the right kind of context—so that scalability is maintained and precision is enhanced. However, no better alternative exists, if higher precision is desired. All other standard precision enhancements (e.g., flow sensitivity, more precise heap modeling with shape analysis) typically either fail to improve precision or are impossible to scale, for a language like Java.

For instance, for a 2-type-sensitive analysis with a 1-context-sensitive heap, we would have $C = T \times T$, $HC = T$ and constructors:

Record (*heap, ctx*) = *first(ctx)*
Merge (*heap, hctx, invo, ctx*) = *pair*($\mathcal{C}_{\mathcal{A}}$*(heap), hctx)*

where $\mathcal{C}_{\mathcal{A}} : H \rightarrow T$ is an auxiliary function mapping each heap abstraction to the class containing the allocation. The type-sensitive analysis is otherwise isomorphic to an object-sensitive one: any type that appears in a context has to be the class containing the allocation site that would appear in the same context position.

Another well-known context-sensitivity approach is that of the *Cartesian Product Algorithm* [Agesen, 1995]. The algorithm treats as context of a method call the abstract values of all parameters to the method call, including the receiver object and actual arguments. This approach has not yet been found to yield useful scalability/precision tradeoffs for pointer analysis, although it has had application in less expensive analyses (e.g., type inference).

Several more flexible combinations of contexts are possible. An *introspective analysis* [Smaragdakis et al., 2014] adjusts its context per program site, based on easy-to-compute statistics from a context-insensitive analysis run. (This kind of context adjustment, or *refinement* is more common in demand-driven analyses, discussed in Section 5.3.) A relative balance between call-site sensitivity and object sensitivity has been explored with *hybrid analysis* [Kastrinis and Smaragdakis, 2013b]. A hybrid analysis models separately static and virtual method calls (input relations VCall and SCall, from Chapter 3, in our formalism), favoring object sensitivity for the former and call-site sensitivity for the latter. For analyses with more than one element of context, the design space becomes large, with interesting choices at every step—for instance, how many elements of context should be call sites vs. allocation sites for the analysis of a static method called inside a virtual method? With appropriate tuning of this form, hybrid analyses have experimentally yielded superior tradeoffs of precision and performance.

Complexity. An important note for the area of context-sensitive pointer analysis concerns the subtlety of definitions and the need for careful statement of claims and results. This becomes quite evident in the existence of "paradoxes" often arising in theoretical analyses. Horwitz [1997] has proven that precise context-sensitive (but *flow-insensitive*—i.e., not taking into account the order of statements; see Chapter 5) analysis is $\mathcal{NP}$-hard. Yet the analysis considered is virtually identical to our Andersen-style pointer analysis, expressed in Datalog—a language with guaranteed polynomial execution. The subtle difference concerns the input language. Horwitz's approach assumes an intermediate language with arbitrary levels of pointer dereferences. That is, the analysis considers a statement such as "`*a = **a;`" to be a single instruction, whereas our load/store intermediate language would break the statement up into three instructions:

```
t = *a;
t2 = *t;
*a = t2;
```

The difference is that, in our case, the three instructions are not treated as a unit (i.e., their order and contiguity are not guaranteed), unlike in the Horwitz result. Treating complex dereferences as a unit lends a certain amount of *flow sensitivity* to the analysis, which is sufficient for establishing the $\mathcal{NP}$-hardness result. Rinetzky et al. [2008] study the impact of various forms of limited flow-sensitivity in the complexity of must- and may-alias analysis and they use the term *partially flow-sensitive* for the above representation.

A similar recent "paradox" arose from the proof by Van Horn and Mairson [2008] that the kCFA algorithm [Shivers, 1991] is $\mathcal{EXPTIME}$-complete, for $k \geq 1$. Yet the kCFA algorithm is one of the prototypical call-site-sensitive analyses, seemingly forming the basis for multiple implementations guaranteed to run in polynomial time, including our earlier Datalog formulation. The difference, again, turns out to be in the input language. As Might et al. [2010] show, if we assume a functional language, with automatic closure of environment variables, the abstract-interpretation-based kCFA algorithm has an exponential value space: although the number of different contexts is polynomial, the

number of possible values of the environment is not. An analyzed environment can have different combinations of closed values for variables (depending on prior program actions in the abstract interpretation) for the same context. In this way, a single call may need to be analyzed for a large number of states (exponential in the number of variables in the program text). In object-oriented languages, with explicit copying of variable values at every call site, no such combinatorial explosion arises: each copied variable acquires the abstract values that the analysis has computed for it so far, without concern for the values of other variables. As a result, the algorithm loses some precision but remains polynomial, as our Datalog implementation suggests.

5

Flow Sensitivity, Must-Analysis, and ... Pointers

Our presentation so far has omitted several interesting topics in the pointer analysis area. These are tangential to the main emphasis of earlier chapters, but important in their own, heterogeneous, respects.

5.1 Flow Sensitivity

Flow sensitivity refers to the ability of an analysis to take control flow into account when analyzing a program. Consider a simple program fragment:

```
a = new A1();
if (condition()) {
   a = new A2();
   c.f = a;
}
```

Will the analysis treat the program statements in order, and with an understanding of the branching connectives? Thus, will, the `f` field of the object pointed by `c` be assigned the abstract object `new A1()` (an infeasible assignment, based on control flow) or just the object `new A2()`?

In the previous chapters, we have implicitly taken a *flow-insensitive* approach: we treat all instructions as an unordered set of statements that induce constraints, regardless of their position in the program. Thus, we can view the above program fragment as an unordered set consisting of two ALLOC and one STORE instructions. The result is to consider the `new A1()` object to flow to the `f` field of whichever object `c` points to.

However, in reality, adding flow sensitivity to the points-to analysis of our earlier chapters is an orthogonal matter, addressed by mere preprocessing. If we put the input program in a *static single assignment (SSA)* form, our standard analysis rules yield a flow-sensitive analysis.[1] An SSA representation of the program assigns exactly once to each variable and merges the matching temporary variables from different program branches via the use of ϕ ("phi") statements. ϕ statements are indicators of the merging of control flow and each analysis can treat them differently. In our case, an SSA transformation of the input, with ϕ statements treated as successive MOVE instructions, yields a flow-sensitive analysis. Consider the SSA-transformed version of our earlier example:

```
a1 = new A1();
if (condition()) {
   a2 = new A2();
   c.f = a2;
}
// a3 = phi(a1,a2)
a3 = a1;
a3 = a2;
```

(Although we speak of an SSA form, the `a3` variable is assigned twice, as a result of the translation of the special ϕ node. However, the assignments to `a3` are only virtually present, for the purposes of our pointer analysis, as formulated earlier. We could equivalently add a *phi* instruction in our intermediate language—as in the next section—and treat it similarly to a MOVE instruction in earlier rules.)

[1] Albeit, without a flow-sensitive object model/heap model. That is, the points-to sets of local variables are distinguished per program branch, but the points to sets of objects (relation FLDPOINTSTO in our model) are not.

The two different branches are now cleanly separated. The program is represented by two ALLOC instructions, two MOVE instructions and one LOAD. After the renaming, our analysis no longer confuses the flow of the two abstract objects, although it has no concept of ordering of statements.

Our analysis, as formulated earlier, can acquire flow sensitivity by a mere SSA transformation of the input because it never keeps information per-program-point. In contrast, other analyses do, and require special handling of control-flow constructs. A representative example is a must-point-to analysis.

5.2 Must-Analysis

The pointer analysis of previous chapters is a *may-analysis*: it produces an over-approximation of the fully precise result. The analysis may freely infer points-to relationships that will not hold in real executions, in addition to the ones that will. In contrast, several clients of pointer analysis information need absolute certainty regarding the existence of a certain points-to relationship—examples include intrusive program refactoring or optimization, high-quality error reporting, etc. This certainty requires a *must-analysis*: an under-approximation of program behavior, inferring results only when it is certain that they will hold during program execution.

The *must-* vs. *may-* flavor of an analysis refers to the intent of the analysis design. A *sound* analysis achieves this intent, i.e., is truly under- or over-approximate as desired. Thus, *soundness* for a must-analysis means avoiding spurious inferences, whereas soundness for a may-analysis means not missing true inferences.

The topic of *must-* vs. *may-*analysis extends more generally than just pointer analysis, to the entire area of static analysis. It is useful to consistently assign the meaning of under- vs. over-approximate analysis to the "must" and "may" prefixes, respectively, instead of relying on intuitive understanding of the English terms. For example, negating the result of a may-point-to analysis (i.e., taking the complement of its output) yields a must-not-point-to analysis (and *not* a may-not-point-

to analysis): the complement of an over-approximation is an under-approximation of the precise result.

Applied specifically to pointer analysis, the practical issue is that common uses of the analysis results subtly introduce negation, thus violating the over-approximate nature of the analysis. That is, using the results of a may-point-to analysis as if they were fully precise will often violate soundness of the client. For instance, when asking the question "can a heap object field be un-initialized?", we typically desire an over-approximate answer, i.e., a may-analysis for un-initialized fields. That is, if the field can possibly be un-initialized in some real program execution, the analysis should issue a warning about the field. However, a may-point-to analysis does not give us enough information to compute this answer! The may-point-to analysis will yield a superset of the values assigned to the field. The straightforward definition of an un-initialized field analysis is to then report as un-initialized the fields that do not have any values assigned to them. However, this is an under-approximate result: the field may appear initialized merely because of the lack of precision of the may-point-to analysis, which conflates several actual program executions if these map to the same context or flow abstraction of the may-point-to analysis.

A must-point-to analysis can address this need. We can express such an analysis in our usual executable formalism, as Datalog inference rules. A realistic must-analysis has significant complexity and requires careful modeling of language semantics to ensure soundness. In contrast, our model is a simplistic must-analysis to serve as a point of reference in our illustration of important concepts. This is unlike our earlier may-analyses, which closely model perfectly realistic, highly effective implementations.

Figure 5.1 shows our input relations for a must-analysis. These are quite similar to our earlier relations: VCall remains the same, while Alloc, Move, Load and Store merely acquire a new argument identifying the program instruction. We assume an SSA representation, therefore the assigned variables of Alloc and Move are also uniquely identified. The meaningful new additions are Phi and Next relations. Phi adds a ϕ instruction to our intermediate language. For simplicity,

our PHI only merges two data values, instead of an arbitrary number. The NEXT relation expresses directed edges in the control-flow graph (CFG): NEXT(*i,j*) means that i is a CFG predecessor of j.

Figure 5.1 also shows the computed relations of our must-point-to analysis. These relations are independent of earlier may-point-to relations (e.g., Figures 2.1 or 4.1): the must-analysis may run independently of the may-analysis, and the results of the two can be combined in a variety of ways.

The two relations computed are MUSTPOINTTO and FLDMUSTPOINTTO, by analogy to the respective concepts in the may-analysis. MUSTPOINTTO(*var,heap*) means that variable *var* shall point to the most recently allocated object of allocation site *heap*, *modulo recursion*, i.e., the most recent for the same activation of the enclosing method. Thus, the abstract object *heap* represents a single run-time object, and can be treated as such (e.g., writes to its fields will replace earlier values—a concept commonly called *strong update*). Similarly, FLDMUSTPOINTTO(*instr, baseH, fld, heap*) signifies that, at the program point immediately after instruction *instr*, the most recently allocated object of site *baseH* always points via its *fld* field to the most recently allocated object of site *heap* (with "more recent" always defined modulo recursion—we discuss the significance of this later).

ALLOC(*i : I, var : V, heap : H, inMeth : M*)	*# i: var = new ...*
MOVE(*i : I, to : V, from : V*)	*# i: to = from*
LOAD(*i : I, to : V, base : V, fld : F*)	*# i: to = base.fld*
STORE(*i : I, base : V, fld : F, from : V*)	*# i: base.fld = from*
VCALL(*base: V, sig: S, invo: I, inMeth: M*)	*# invo: base.sig(..)*
PHI(*i: I, to : V, from1: V, from2: V*)	*# i: to = ϕ(from1,from2)*
NEXT(*i : I, j : I*)	*# j is CFG successor of i*
MUSTPOINTTO(*var : V, heap : H*)	
FLDMUSTPOINTTO(*instr : I, baseH : H, fld : F, heap : H*)	

Figure 5.1: Input and computed relations for must-analysis.

Figure 5.2 presents our model of a must-point-to analysis. For economy of expression, we introduce FORALL: syntactic sugar that hides a Datalog pattern for enumerating all members of a set and ensuring that a condition holds universally. An expression "FORALL(i): P(i) $\rightarrow$ Q(i,...)" is true if Q(i,...) holds for all i such that P(i) holds. Such an expression can be used in a rule body, as a condition for the rule's firing.

The first five rules are direct counterparts of the may-analysis rules from Figure 2.2. All must-point-to inference starts at the point of an object allocation and propagates through the program. The last two rules are more interesting. They show how a model of the heap, captured by relation FLDMUSTPOINTTO, can propagate from one instruction to the next. The first of these rules states that if at all predecessors, i, of an instruction j, a heap object, *baseH*, is known to point to another heap object, *heap*, via field *fld*, then the same inference holds at instruction j, *provided* that j is not a store to field *fld*, a method call, or an allocation site of one of the two involved objects, *baseH* and *heap*.

The final rule is similar: it propagates the same information from instruction i to j, if instruction j is a STORE to field *fld* but the analysis has established that the base variable of the STORE must point to an object different from *baseH*.

The rules give a glimpse of the difficulties involved in specifying a correct must-point-to analysis, as well as of the possibilities. The analysis model shown leaves significant room for improvement, in several different ways:

- The analysis is quite conservative in that it stops the propagation of its heap model on every method call or store to the same field. In practice, this is typically alleviated by running the must-analysis after a may-point-to analysis and call-graph computation (e.g., see [Sridharan et al., 2013]). The latter computes an over-approximation of the (abstract) objects a base variable can point to and of the methods possibly invoked at a call site. This can be used to make the filtering of the last two rules in Figure 5.2 much more precise, allowing more inferences. For instance, the use of MUSTPOINTTO in the last rule can be replaced by a

```
MustPointTo(var, heap) ←
  Alloc(_, var, heap, _).

MustPointTo(to, heap) ←
  Move(_, to, from), MustPointTo(from, heap).

MustPointTo(to, heap) ←
  Phi(_, to, from1, from2),
  MustPointTo(from1, heap), MustPointTo(from2, heap).

FldMustPointTo(i, baseH, fld, heap) ←
  Store(i, base, fld, from),
  MustPointTo(from, heap), MustPointTo(base, baseH).

MustPointTo(to, heap) ←
  Load(i, to, base, fld), MustPointTo(base, baseH),
  FldMustPointTo(i, baseH, fld, heap).


FldMustPointTo(j, baseH, fld, heap) ←
  (forall(i): Next(i,j) →
    FldMustPointTo(i, baseH, fld, heap),
    !Store(j, _, fld, _),
    !VCall(_, _, j, _),
    !Alloc(j, _, baseH),
    !Alloc(j, _, heap)).

FldMustPointTo(j, baseH, fld, heap) ←
  (forall(i): Next(i,j) →
    FldMustPointTo(i, baseH, fld, heap),
    Store(j, base, fld, _),
    MustPointTo(base, baseH2), baseH2 != baseH).
```

Figure 5.2: Datalog rules for a simple must-point-to analysis.

!VarPointsTo: the logic would effectively change from "must-point-to a different base" to "not-may-point-to the same base". Similarly, in the next-to-last rule, the calls that could potentially result in a write to field *fld* can be identified and used as a filter, instead of the generic VCall(_, _, *j*, _).

- The rule handling Store instructions is draconian, with its premise nearly impossible to establish. The analysis needs to know with certainty which objects both variable *base* and variable *from* must point to. Therefore, very few FldMustPointTo inferences can be made with this analysis model. Recall that must-point-to information refers to *concrete* objects: the most recent instance allocated by a given Alloc site. Instead, realistic must-analyses often also represent *summary* objects and track their inter-relations. A general way for doing so is via access paths. For instance, the analysis can track which field expressions, `base.fld`, in the program text, are aliases, or which concrete object they must point to, without establishing which concrete object variable `base` must point to.

- Most importantly, the analysis has no handling of call instructions. Therefore, it remains strictly intra-procedural. In order to handle method calls, a must-analysis needs to become context sensitive. For a straightforward approach, in order to propagate heap models at a call site, the context used to analyze the callee needs to uniquely identify the call site: no two control flows can be conflated. This precludes the use of "novel" context abstractions, as in object sensitivity and type sensitivity (Chapter 4). Generally, the role of context in a must-analysis is quite different from that in a may-analysis. For instance, a deeper context allows a must-analysis to infer more facts, instead of fewer. Furthermore, a context-insensitive fact typically has a "forall contexts" meaning in a must-analysis, i.e., implies its context-sensitive counterpart. In a may-analysis the implication is inverse.

- The correctness of the rules, and even the form of predicate MustPointTo, subtly depends on our assumptions. The inter-

mediate language is in SSA form, hence every variable has a single point of assignment. Thus, MustPointTo does not need to include an instruction in its arguments, unlike predicate FldMustPointTo: MustPointTo facts always hold once established, since there are no further assignments.

Furthermore, recall that MustPointTo(*var,heap*) means that variable *var* can only point to the most recently allocated object of allocation site *heap*, modulo recursion. The "modulo recursion" qualification is important. If MustPointTo were to track the true most recently allocated object of site *heap*, then, even without assignments, a MustPointTo fact could be later invalidated: a method call can (eventually) result in the original method being invoked and site *heap* producing a new object. Since our analysis is strictly intra-procedural, there is no need to track the true most recently allocated object for an allocation site. If, however, we were to propagate FldMustPointTo facts inter-procedurally, we would need such information, which would likely require tracking MustPointTo facts per-instruction.

Generally, must-analyses can vary greatly in sophistication and can be employed in an array of different combinations with may-analyses. The analysis of Balakrishnan and Reps [2006], which introduces the *recency abstraction*, distinguishes between the most recently allocated object at an allocation site (a concrete object, allowing strong updates) and earlier-allocated objects (represented as a summary node). The analysis additionally keeps information on the size of the set of objects represented by a summary node. At the extreme, one can find full-blown shape analysis approaches, such as that of Sagiv et al. [2002], which explicitly maintains must- and may- information simultaneously, by means of three-valued truth values: a relationship can definitely hold ("must"), definitely not hold ("must not", i.e., negation of "may"), or possibly hold ("may"). Summary and concrete nodes are again used to represent knowledge, albeit in full detail, as captured by arbitrary predicates whose value is maintained across program statements, at the cost of a super-exponential worst-case complexity.

5.3 Other Directions

Several more topics in pointer analysis research have attracted significant interest and represent promising areas for future work or connections to neighboring fields. We summarize them next, for further reference.

CFL reachability formulation. In this tutorial, we formulated pointer analysis algorithms in Datalog. Employing a restricted language not only offers guarantees of termination and complexity bounds, but also permits more aggressive optimization of the language features.

Along these lines, pointer analysis and other related analyses have been formulated as a *context-free language (CFL) reachability* problem. The idea is that we may encode an input program as a labeled graph, and a specific analysis corresponds to the definition of a context-free grammar, G. The relation being computed holds for two nodes of the graph if and only if there exists a path from one node to the other, such that the concatenation of the labels of edges along the path belongs in the language $L(G)$ defined by G.

Specifically, the input graph consists of nodes that are program expressions and edges represent relations between those expressions. For instance, an edge may encode a field access (load/store), a method invocation, a pointer dereference, etc. The exact choice of domains depends on the specific analysis being run and the problem it addresses. Since we want to express many input relations, we need many types of edges, represented as labels (e.g., we can label a field access edge by the name of the field). For a given analysis, a context-free grammar G encodes the desired computed relations (e.g., which pointers are memory aliases) as non-terminal symbols, and supplies derivation rules that express how they relate to the simpler relations represented by graph edges. The CFL reachability answer is then commonly computed by employing a dynamic programming algorithm.

The first application of such a framework in program analysis was designed to solve various interprocedural dataflow-analysis problems [Reps et al., 1995], but CFL reachability has since been used in a wide range of problems, such as: (i) the computation of points-to relations

[Reps, 1998], (ii) the (demand-driven) computation of may-alias pairs for a C-like language [Zheng and Rugina, 2008], (iii) Andersen-style pointer analysis for Java [Sridharan et al., 2005].

Any CFL reachability problem can be converted to a Datalog program [Reps, 1998], but the inverse is not true (i.e., CFL reachability corresponds to a restricted class of Datalog programs, the so-called "chain programs"). Thus, the primary advantage of CFL reachability is that it permits more efficient implementations.

An even more restrictive variant, *Dyck-CFL reachability*, can be obtained by restricting the underlying CFL to a Dyck language, i.e., one that generates balanced-parentheses expressions. Although restrictive, this approach still suffices for some simple pointer analysis algorithms, while allowing very aggressive optimization, often with impressive performance gains [Zhang et al., 2013].

Data-flow analysis. Data-flow analysis is a program analysis framework, widely employed in traditional compiler design. The essence of data-flow analysis is to maintain information on properties of data values per-program-point and to propagate this information by defining monotonic *transfer functions* (determining how information is propagated between neighboring statements inside a basic block) and *confluence operators* (determining how information is propagated at basic block boundaries, where control flow is possibly merged or split). An implementation applies these propagation operators to change the computed values over all program statements until they reach a fixpoint.

Data-flow analysis is used for the computation of many intraprocedural properties in compilers, such as variable liveness, reaching definitions, constantness of values, and more. With appropriate summarization techniques, data-flow results can also be propagated interprocedurally.

Pointer analysis, as described in this tutorial, is naturally interprocedural and the implementation model we examined, based on the Datalog language, can be viewed as a generalization of traditional data-flow equations. (Indeed, modern compiler textbooks [Aho et al., 2006, 12.3] explicitly refer to the Datalog approach as such a generalization

of data-flow analysis.) Whereas data-flow analysis typically maintains sets of values (each corresponding to a program element), a Datalog-based declarative analysis approach populates arbitrary relations over many values. Instead of fixing a single pattern of recursive definitions (transfer functions and confluence operators), full freedom is afforded in defining relations recursively.

Conversely, a flow-sensitive Datalog-based analysis typically employs precisely the same concepts as a standard data-flow analysis for linking information across intra-procedural program points.

Constraint graph approaches and optimizations. Several optimization techniques have appeared in the pointer analysis literature, based on the concept of the *constraint graph*: a graph whose nodes denote pointer expressions and whose edges denote flow between these pointer expressions.

Such an edge may arise either: (i) *before* points-to computation—due to an explicit assignment instruction "$q = p$", for example, that the input program contains—or (ii) *during* points-to computation, by also taking field accesses (i.e., load/store instructions) into account in order to infer additional flow. These techniques have typically targeted the C language, where the instructions of interest are pointer dereferences (analogous to field accesses) and also address-of (&) operations (analogous to assignments of heap allocations).

A variety of constraint graph optimization techniques have been presented that can be applied either *offline* (i.e., before points-to computation) [Hardekopf and Lin, 2007b, Rountev and Chandra, 2000], or *online* (i.e., during points-to computation) [Fähndrich et al., 1998]. Hybrid approaches also exist [Hardekopf and Lin, 2007a]. The essence of most of these techniques lies in identifying nodes with guaranteed-equivalent points-to sets and collapsing them into a single representative node. Such equivalence classes may arise, for instance, when nodes participate in a cycle of the constraint graph [Fähndrich et al., 1998, Heintze and Tardieu, 2001a], or even if they share a common dominator [Nasre, 2012]. The *set-based pre-processing* technique [Smaragdakis et al., 2013] generalizes such approaches by also allowing the removal of

edges (and not just merging of nodes) from the constraint graph. It also restricts the application of optimizations to an intra-procedural setting, so that they can be applied in conjunction with nearly any pointer analysis algorithm, together with on-the-fly call-graph construction.

Binary Decision Diagrams. A promising direction in the implementation of points-to analysis algorithms consists of the use of ordered *binary decision diagrams (BDDs)* for representing relations. A BDD is a data structure used to encode a Boolean predicate. Effectively it is a binary decision tree that tries to merge isomorphic subtrees, and can thus serve as a compressed normalized representation of a relation. A more detailed description of BDDs and their use for storing pointer analysis relations is given by Berndl et al. [2003].

A BDD-based implementation is a good fit for a relational view of the analysis. Therefore, Datalog can be used as a high-level declarative language for writing analyses, with BDDs used in the implementation of relations inside the underlying Datalog engine. This is the approach of the `bddbddb` system [Whaley and Lam, 2004, Whaley et al., 2005]. Other approaches, such as the PADDLE framework [Lhoták and Hendren, 2008, Lhoták, 2006], use a specialized low-level relational language, backed by a BDD-based implementation. The analysis itself is expressed in imperative code so that it becomes easier to interact with external components.

BDDs have been hugely beneficial in other areas, such as model checking. For pointer analysis, however, the record is mixed. The performance of a BDD-based approach is very sensitive to the exact choice of decision variable ordering. In fact, previous approaches have experimented heavily in identifying an efficient ordering that minimizes the size of the BDDs [Berndl et al., 2003]. However, finding orderings that perform well for all of the crucial BDDs in a framework with many interconnected analyses remains a challenging task that also hurts the framework's maintainability significantly; it requires a "trial and error" process of exploring a huge space of possibilities [Lhoták, 2006, Whaley et al., 2005]. Such optimization is also often detrimental for other parts of the analysis: the overall "best" variable ordering may be severely

sub-optimal for some analysis relations (while optimizing others). As a result, non-BDD-based implementations of identical analyses have often exhibited significant performance improvements, if care is taken to eliminate space-redundancies in the analysis [Bravenboer and Smaragdakis, 2009a]. It remains to be seen whether future BDD-based implementations can provide the space economy and speed of well-ordered BDD operations without suffering great costs.

Incrementality and summaries. The analyses we have considered so far have been *all-points*, *client-agnostic* analyses. In contrast, *demand-driven* pointer analysis computes only those results that are necessary for a query at a given program location. *Client-driven* pointer analysis takes into account the precision needs of a specific client. Such analyses have attracted research interest (e.g., [Sridharan et al., 2005, Zheng and Rugina, 2008, Heintze and Tardieu, 2001b]). The question is algorithmically interesting, because the base algorithms for pointer analysis (e.g., our formulation of an Andersen-style analysis) are not easily incrementalizable. However, different techniques in this space have achieved significant scalability benefits via approaches that take into account the needs of clients or specific program points and automatically adjust the precision (e.g., context- or field-sensitivity) of an analysis [Guyer and Lin, 2003, Sridharan and Bodík, 2006, Liang and Naik, 2011].

Another direction to analysis incrementality is *modular* pointer analysis [Cheng and Hwu, 2000, Wilson and Lam, 1995]. Modular analysis computes a summary of the behavior of a function or method and analyzes the rest of the application against this summary. Well-known techniques include that of Wilson and Lam [1995], which computes multiple summaries per function and employs memoization of the data-flow solution for a particular context. Chatterjee et al. [1999] compute a single summary per function, based on its global effects. Techniques for computing summaries are an active research topic for more than pointer analysis [Dillig et al., 2008], and represent a significant future challenge.

Correctness and Abstract Interpretation. In this tutorial we ignored issues of formally establishing the correctness of a pointer analysis. This topic is also directly related to *abstract interpretation*: a widely-used framework for specifying static analyses [Cousot and Cousot, 1977, 1979]. Abstract interpretation consists of specifying analyses by abstracting over the concrete semantics of a programming language. That is, given a formally specified language, with concrete semantics for all language features, an abstract interpretation-based analysis provides an alternative, abstract semantics over the same language. The key to the approach is that the two semantics are linked. The abstract and concrete value domains are connected via abstraction and concretization functions. Under the right conditions for these functions relative to the respective semantics, several correctness results can be derived. For instance, the soundness of an over-approximate analysis is established if a transition from abstract state s to abstract state t in the abstract semantics models all transitions in the concrete semantics, i.e., all concrete states abstracting to s can only transition to concrete states abstracting to t.

Abstract interpretation is a very general framework, capturing a wealth of practical static analyses. It is also the foremost way to formally establish the correctness of an analysis. The approach still requires effort, even for languages that have full formal specifications. In order to easily link the concrete and abstract semantics, the concrete semantics of the language is often re-stated, in terms of convenient domains and transition functions. As a result, the obligation is shifted from proving that the analysis over-approximates concrete execution to proving that the alternative concrete semantics is equivalent to the original—a typically easier task.

At the same time, one should employ the framework carefully or risk a reduced understanding of the properties of the static analysis. The analysis can occasionally become too expensive because it is allowed to execute over the analyzed program in a domain that (although finite) may still model elements such as precise (abstract) execution stacks or detailed heap images. One such instance is the discovery of Van

Horn and Mairson [2008] that the kCFA algorithm [Shivers, 1991] is $\mathcal{EXPTIME}$-complete for $k \geq 1$, as mentioned in Chapter 4.

6

Conclusions

In this monograph, we surveyed the pointer analysis area in tutorial form. Pointer analysis is a mature area with a rich background and intense research interest. Recent years have seen an extension of the set of clients for the analysis—from traditional compilers to program understanding, refactoring, verification, and bug detection tools. Furthermore, programming trends have made pointer analysis crucial: recent mainstream languages are increasingly heap-intensive—the emphasis has shifted from C-like to object-oriented languages, such as Java and C#, and to scripting languages, such as JavaScript, Python, and Ruby.

If pointer analysis is a scalable, well-understood approach, one may rightly wonder why it has not yet been widely applied in language processing mechanisms. Although several tools employ pointer analysis algorithms, the techniques we discussed have seen limited use in the best known compilers, IDEs, or runtime systems.

The reason largely has to do with the advent of more modular compilation and dynamic language features. Pointer analysis approaches have made great strides but still typically require whole-program availability and cannot deal soundly with dynamic features. Thus, the next set of challenges for pointer analysis do not have to do with scalability

and precision in a controlled setting but with applicability under the environmental conditions of realistic languages.

If research progress addresses these challenges, we expect that the use of pointer analysis will become universal in modern programming tools, lending the analysis huge practical importance for future programming tasks.

Acknowledgments

We thank Martin Bravenboer who originated the development of several declarative analysis concepts and algorithms in the Doop framework, as well as George Kastrinis for Doop contributions. We gratefully acknowledge funding by the European Research Council via a Starting/Consolidator grant (Spade).

References

Ole Agesen. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *Proc. of the 9th European Conf. on Object-Oriented Programming*, ECOOP '95, pages 2–26. Springer, 1995. ISBN 3-540-60160-0.

Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006. ISBN 0321486811.

Lars O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.

David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. In *Proc. of the 11th Annual ACM SIGPLAN Conf. on Object Oriented Programming, Systems, Languages, and Applications*, OOPSLA '96, pages 324–341, New York, NY, USA, 1996. ACM. ISBN 0-89791-788-X.

Gogul Balakrishnan and Thomas W. Reps. Recency-abstraction for heap-allocated storage. In *Proc. of the 14th International Symp. on Static Analysis*, SAS '06, pages 221–239. Springer, 2006.

Marc Berndl, Ondrej Lhoták, Feng Qian, Laurie J. Hendren, and Navindra Umanee. Points-to analysis using BDDs. In *Proc. of the 2003 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI '03, pages 103–114, New York, NY, USA, 2003. ACM. ISBN 1-58113-662-5.

Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proc. of the 24th Annual ACM SIGPLAN Conf. on Object Oriented Programming, Systems, Languages, and Applications*, OOPSLA '09, New York, NY, USA, 2009a. ACM. ISBN 978-1-60558-766-0.

Martin Bravenboer and Yannis Smaragdakis. Exception analysis and points-to analysis: Better together. In *Proc. of the 18th International Symp. on Software Testing and Analysis*, ISSTA '09, pages 1–12, New York, NY, USA, 2009b. ACM. ISBN 978-1-60558-338-9.

Ramkrishna Chatterjee, Barbara G. Ryder, and William A. Landi. Relevant context inference. In *Proc. of the 26th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, POPL '99, pages 133–146. ACM, 1999. ISBN 1-58113-095-3.

Ben-Chung Cheng and Wen-Mei W. Hwu. Modular interprocedural pointer analysis using access paths: Design, implementation, and evaluation. In *Proc. of the 2000 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI '00, pages 57–69, New York, NY, USA, 2000. ACM. ISBN 1-58113-199-2.

Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *POPL '79: Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 269–282, 1979.

Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proc. of the 9th European Conf. on Object-Oriented Programming*, ECOOP '95, pages 77–101. Springer, 1995. ISBN 3-540-60160-0.

Isil Dillig, Thomas Dillig, and Alex Aiken. Sound, complete and scalable path-sensitive analysis. In *Proc. of the 2008 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI '08, pages 270–280, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-860-2.

Michael Eichberg, Sven Kloppenburg, Karl Klose, and Mira Mezini. Defining and continuous checking of structural program dependencies. In *Proc. of the 30th International Conf. on Software Engineering*, ICSE '08, pages 391–400, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-079-1.

Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proc. of the 1994 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI '94, pages 242–256, New York, NY, USA, 1994. ACM. ISBN 0-89791-662-X.

Manuel Fähndrich, Jeffrey S. Foster, Zhendong Su, and Alexander Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Proc. of the 1998 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI '98, pages 85–96, New York, NY, USA, 1998. ACM. ISBN 0-89791-987-4.

Chen Fu and Barbara G. Ryder. Exception-chain analysis: Revealing exception handling architecture in Java server applications. In *Proc. of the 29th International Conf. on Software Engineering*, ICSE '07, pages 230–239, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2828-7.

Samuel Z. Guyer and Calvin Lin. Client-driven pointer analysis. In *Proc. of the 10th International Symp. on Static Analysis*, SAS '03, pages 214–236. Springer, 2003. ISBN 3-540-40325-6.

Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. CodeQuest: Scalable source code queries with Datalog. In *Proc. of the 20th European Conf. on Object-Oriented Programming*, ECOOP '06, pages 2–27. Springer, 2006. ISBN 978-3-540-35726-2.

Ben Hardekopf and Calvin Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *Proc. of the 2007 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI '07, pages 290–299, New York, NY, USA, 2007a. ACM. ISBN 978-1-59593-633-2.

Ben Hardekopf and Calvin Lin. Exploiting pointer and location equivalence to optimize pointer analysis. In *Proc. of the 14th International Symp. on Static Analysis*, SAS '07, pages 265–280. Springer, 2007b. ISBN 978-3-540-74060-5.

Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. In *Proc. of the 2001 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI '01, pages 254–263, New York, NY, USA, 2001a. ACM. ISBN 1-58113-414-2.

Nevin Heintze and Olivier Tardieu. Demand-driven pointer analysis. In *Proc. of the 2001 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI '01, pages 24–34, New York, NY, USA, 2001b. ACM. ISBN 1-58113-414-2.

Michael Hind. Pointer analysis: haven't we solved this problem yet? In *Proc. of the 3rd ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '01, pages 54–61, New York, NY, USA, 2001. ACM. ISBN 1-58113-413-4.

Susan Horwitz. Precise flow-insensitive may-alias analysis is NP-hard. *ACM Trans. Program. Lang. Syst.*, 19(1):1–6, January 1997. ISSN 0164-0925.

Neil Immerman. *Descriptive Complexity*. Graduate texts in computer science. Springer, 1999. ISBN 978-1-4612-6809-3.

Vini Kanvar and Uday P. Khedker. Heap abstractions for static analysis. *CoRR*, abs/1403.4910, 2014. URL `http://arxiv.org/abs/1403.4910`.

George Kastrinis and Yannis Smaragdakis. Efficient and effective handling of exceptions in Java points-to analysis. In *Proc. of the 22nd International Conf. on Compiler Construction*, CC '13, pages 41–60. Springer, 2013a. ISBN 978-3-642-37050-2.

George Kastrinis and Yannis Smaragdakis. Hybrid context-sensitivity for points-to analysis. In *Proc. of the 2013 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI '13, New York, NY, USA, 2013b. ACM. ISBN 978-1-4503-2014-6.

Monica S. Lam, John Whaley, V. Benjamin Livshits, Michael C. Martin, Dzintars Avots, Michael Carbin, and Christopher Unkel. Context-sensitive program analysis as database queries. In *Proc. of the 24th Symp. on Principles of Database Systems*, PODS '05, pages 1–12, New York, NY, USA, 2005. ACM. ISBN 1-59593-062-0.

William Landi. Undecidability of static analysis. *LOPLAS*, 1(4):323–337, 1992.

William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural aliasing. In *Proc. of the 1992 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI '92, pages 235–248, New York, NY, USA, 1992. ACM. ISBN 0-89791-475-9.

Ondřej Lhoták. *Program Analysis using Binary Decision Diagrams*. PhD thesis, McGill University, January 2006.

Ondřej Lhoták and Laurie J. Hendren. Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation. *ACM Trans. Softw. Eng. Methodol.*, 18(1):1–53, 2008. ISSN 1049-331X.

Yue Li, Tian Tan, Yulei Sui, and Jingling Xue. Self-inferencing reflection resolution for Java. In *Proc. of the 28th European Conf. on Object-Oriented Programming*, ECOOP '14, pages 27–53. Springer, 2014. ISBN 978-3-662-44201-2.

Percy Liang and Mayur Naik. Scaling abstraction refinement via pruning. In *Proc. of the 2011 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI '11, pages 590–601, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0663-8.

Benjamin Livshits. *Improving Software Security with Precise Static and Runtime Analysis*. PhD thesis, Stanford University, December 2006.

Benjamin Livshits, John Whaley, and Monica S. Lam. Reflection analysis for Java. In *Proc. of the 3rd Asian Symp. on Programming Languages and Systems*, APLAS '05, pages 139–160. Springer, 2005. ISBN 3-540-29735-9.

Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: A manifesto. *Commun. ACM*, 58(2):44–46, January 2015. ISSN 0001-0782. . URL `http://doi.acm.org/10.1145/2644805`.

Magnus Madsen, Benjamin Livshits, and Michael Fanning. Practical static analysis of JavaScript applications in the presence of frameworks and libraries. In *Proc. of the ACM SIGSOFT International Symp. on the Foundations of Software Engineering*, FSE '13, pages 499–509. ACM, 2013. ISBN 978-1-4503-2237-9.

Matthew Might, Yannis Smaragdakis, and David Van Horn. Resolving and exploiting the k-CFA paradox: Illuminating functional vs. object-oriented program analysis. In *Proc. of the 2010 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI '10, pages 305–315, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0019-3.

Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *Proc. of the 2002 International Symp. on Software Testing and Analysis*, ISSTA '02, pages 1–11, New York, NY, USA, 2002. ACM. ISBN 1-58113-562-9.

Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Trans. Softw. Eng. Methodol.*, 14(1):1–41, 2005. ISSN 1049-331X.

Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for Java. In *Proc. of the 2006 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI '06, pages 308–319, New York, NY, USA, 2006. ACM. ISBN 1-59593-320-4.

Mayur Naik, Chang-Seo Park, Koushik Sen, and David Gay. Effective static deadlock detection. In *Proc. of the 31st International Conf. on Software Engineering*, ICSE '09, pages 386–396, New York, NY, USA, 2009. ACM. ISBN 978-1-4244-3452-7.

Rupesh Nasre. Exploiting the structure of the constraint graph for efficient points-to analysis. In *Proc. of the 2012 International Symp. on Memory Management*, ISMM '12, pages 121–132. ACM, 2012. ISBN 978-1-4503-1350-6.

Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *Proc. of the 15th International Workshop on Computer Science Logic*, volume 2142 of *CSL '01*, pages 1–19. Springer, 2001. ISBN 3-540-42554-3.

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

Thomas W. Reps. Demand interprocedural program analysis using logic databases. In R. Ramakrishnan, editor, *Applications of Logic Databases*, pages 163–196. Kluwer Academic Publishers, 1994.

Thomas W. Reps. Program analysis via graph reachability. *Information & Software Technology*, 40:701–726, 1998.

Thomas W. Reps. Undecidability of context-sensitive data-independence analysis. *ACM Trans. Program. Lang. Syst.*, 22(1):162–186, 2000.

Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proc. of the 22nd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, POPL '95, pages 49–61, New York, NY, USA, 1995. ACM. ISBN 0-89791-692-1.

John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. of the 17th IEEE Symp. on Logic in Computer Science*, LICS '02, pages 55–74. IEEE Computer Society, 2002. ISBN 0-7695-1483-9.

Noam Rinetzky, Ganesan Ramalingam, Shmuel Sagiv, and Eran Yahav. On the complexity of partially-flow-sensitive alias analysis. *ACM Trans. Program. Lang. Syst.*, 30(3), 2008. .

Atanas Rountev and Satish Chandra. Off-line variable substitution for scaling points-to analysis. In *Proc. of the 2000 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI '00, pages 47–56, New York, NY, USA, 2000. ACM. ISBN 1-58113-199-2.

Barbara G. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. In *Proc. of the 12th International Conf. on Compiler Construction*, CC '03, pages 126–137. Springer, 2003. ISBN 3-540-00904-3.

Mooly Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, May 2002. ISSN 0164-0925.

Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In Steven S. Muchnick and Neil D. Jones, editors, *Program flow analysis: theory and applications*, chapter 7, pages 189–233. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1981. ISBN 0137296819.

Olin Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, may 1991.

Yannis Smaragdakis, Martin Bravenboer, and Ondřej Lhoták. Pick your contexts well: Understanding object-sensitivity. In *Proc. of the 38th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, POPL '11, pages 17–30, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0490-0.

Yannis Smaragdakis, George Balatsouras, and George Kastrinis. Set-based pre-processing for points-to analysis. In *Proc. of the 28th Annual ACM SIGPLAN Conf. on Object Oriented Programming, Systems, Languages, and Applications*, OOPSLA '13, pages 253–270, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2374-1.

Yannis Smaragdakis, George Kastrinis, and George Balatsouras. Introspective analysis: Context-sensitivity, across the board. In *Proc. of the 2014 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI '14, pages 485–495, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2784-8.

Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for Java. In *Proc. of the 2006 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI '06, pages 387–400, New York, NY, USA, 2006. ACM. ISBN 1-59593-320-4.

Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. Demand-driven points-to analysis for Java. In *Proc. of the 20th Annual ACM SIGPLAN Conf. on Object Oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 59–76, New York, NY, USA, 2005. ACM. ISBN 1-59593-031-0.

Manu Sridharan, Satish Chandra, Julian Dolby, Stephen J. Fink, and Eran Yahav. Alias analysis for object-oriented programs. In Dave Clarke, James Noble, and Tobias Wrigstad, editors, *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, volume 7850 of *Lecture Notes in Computer Science*, pages 196–232. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-36945-2.

Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proc. of the 23rd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, POPL '96, pages 32–41, New York, NY, USA, 1996. ACM. ISBN 0-89791-769-3.

Frank Tip and Jens Palsberg. Scalable propagation-based call graph construction algorithms. In *Proc. of the 15th Annual ACM SIGPLAN Conf. on Object Oriented Programming, Systems, Languages, and Applications*, OOPSLA '00, pages 281–293, New York, NY, USA, 2000. ACM. ISBN 1-58113-200-X.

David Van Horn and Harry G. Mairson. Deciding kCFA is complete for EXPTIME. In *Proc. of the 13th ACM SIGPLAN International Conference on Functional programming*, ICFP '08, pages 275–282. ACM, 2008. ISBN 978-1-59593-919-7.

John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proc. of the 2004 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI '04, pages 131–144, New York, NY, USA, 2004. ACM. ISBN 1-58113-807-5.

John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. Using Datalog with binary decision diagrams for program analysis. In *Proc. of the 3rd Asian Symp. on Programming Languages and Systems*, APLAS '05, pages 97–118. Springer, 2005. ISBN 3-540-29735-9.

Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proc. of the 1995 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI '95, pages 1–12, New York, NY, USA, 1995. ACM. ISBN 0-8989791-697-2.

Qirun Zhang, Michael R. Lyu, Hao Yuan, and Zhendong Su. Fast algorithms for Dyck-CFL-reachability with applications to alias analysis. In *Proc. of the 2013 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI '13, pages 435–446, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2014-6.

Xin Zheng and Radu Rugina. Demand-driven alias analysis for C. In *Proc. of the 35th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, POPL '08, pages 197–208, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-689-9.