TLP **16** (5–6): 916–932, 2016. © Cambridge University Press 2016
doi:10.1017/S1471068416000405

*Precise complexity guarantees for pointer analysis via Datalog with extensions**

K. TUNCAY TEKLE and YANHONG A. LIU

Computer Science Department, Stony Brook University, Stony Brook, NY, USA
(*e-mail:* `tuncay,liu@cs.stonybrook.edu`)

submitted 6 May 2016; revised 8 July 2016; accepted 22 August 2016

Abstract

Pointer analysis is a fundamental static program analysis for computing the set of objects that an expression can refer to. Decades of research has gone into developing methods of varying precision and efficiency for pointer analysis for programs that use different language features, but determining precisely how efficient a particular method is has been a challenge in itself.

For programs that use different language features, we consider methods for pointer analysis using Datalog and extensions to Datalog. When the rules are in Datalog, we present the calculation of precise time complexities from the rules using a new algorithm for decomposing rules for obtaining the best complexities. When extensions such as function symbols and universal quantification are used, we describe algorithms for efficiently implementing the extensions and the complexities of the algorithms.

KEYWORDS: Datalog, function symbols, universal quantification, computational complexity, static program analysis, pointer analysis, alias analysis

1 Introduction

Pointer analysis is a static program analysis for computing the set of objects that an expression can refer to. It is a fundamental analysis used for many applications, e.g., debugging (Shapiro and Horwitz 1997), performance analysis (Ghiya *et al.* 2001), dataflow analysis (Shapiro and Horwitz 1997), parallelism (Wilson and Lam 1995; Pearce *et al.* 2007), common subexpression elimination (Diwan *et al.* 1998; Ghiya and Hendren 1998), optimization by incrementalization (Gorbovitski *et al.* 2010), and detection of security vulnerabilities (Avots *et al.* 2005). Consider the following program fragment in an object-oriented programming language:

```
void foo() {
  Object o1 = new Object();
  Object o2;
  if (...) o2 = id(o1);
  else o2 = new Object();
}
void id(Object o) { return o; }
```

* This work was supported in part by NSF under grants CCF-1414078, IIS-1447549, CCF-1248184, CCF-0964196, and ONR under grant N000141512208.

For each variable `o1`, `o2`, and `o`, a pointer analysis may aim to find the set of objects the variable *may* point to. We call each such set a *may-point-to* set. A may-point-to analysis is *sound* if each may-point-to set contains all objects that the variable may point to at runtime. A sound may-point-to analysis is *precise* if no may-point-to set contains more objects than the set of objects that the variable may point to at runtime. Another form of analysis is *must-point-to* analysis, which finds the set of objects that each variable *must* point to. Precise pointer analysis is undecidable (Landi 1992; Ramalingam 1994). Therefore, pointer analysis methods approximate the results, providing different tradeoffs between the precision of the results and the efficiency of the method (Hind and Pioli 2000) while preserving soundness. Whereas may-point-to analysis is an overapproximation, must-point-to analysis is an underapproximation.

From the simple program fragment above, it can be seen that the precision-efficiency tradeoff involves the consideration of control flows, procedures, calling contexts, objects, and other language features such as arrays; leading to a variety of analyses. The worst-case time complexities of existing analyses vary from almost linear (Steensgaard 1996) to doubly exponential (Sagiv *et al.* 1998). However, such worst-case complexities are often not a true indication of analysis time; many researchers provide empirical performance results for their algorithms, and many papers have been written on which pointer analysis one should use, e.g., (Hind and Pioli 2000), including one questioning whether we have solved the pointer analysis problem yet (Hind 2001).

A recent survey (Smaragdakis and Balatsouras 2015) presents existing work on logical specifications of pointer analysis methods in the declarative language *Datalog* and its extensions. Datalog specifications allow expressing the precision aspects of pointer analyses concisely, while ensuring that the analyses are performed in polynomial time, because evaluation of rules in Datalog is guaranteed to be polynomial time. However, just as the worst-case complexities of existing analyses are not a true indication of analysis time, worst-case polynomial time for evaluation of Datalog is not sufficient for understanding actual running times.

In this paper, we consider all the different analyses presented in the survey (Smaragdakis and Balatsouras 2015) expressed in Datalog and its extensions, and study the time complexity of each analysis by using and extending a systematic method for calculating the time complexities for optimal bottom-up evaluation of Datalog rules (Liu and Stoller 2009). To obtain the best complexity, we give a new algorithm for rule decomposition. All analyses require handling rules with many hypotheses, and some require handling extension to Datalog with negation, function symbols, and universal quantification. In each case, we describe the method for handling the extensions and calculating the time complexities. Our methods can be readily used for analyzing other program analyses expressible in Datalog and similar extensions.

2 Language and preliminaries

In this section, we describe Datalog, an optimal method for evaluating a set of Datalog rules with at most two hypotheses each, and a method for calculating the time complexity of the evaluation.

Datalog. *Datalog* is a language for defining rules, facts, and queries, where rules can be used with facts to answer queries. A Datalog rule is of the form:

$$p(a_1, ..., a_k) \leftarrow p_1(a_{11}, ..., a_{1k_1}), ..., \ p_h(a_{h1}, ..., a_{hk_h}).$$

where h is a finite natural number, each p_i (respectively p) is a predicate of finite number k_i (respectively k) arguments, each a_{ij} and a_i is either a constant or a variable, and each variable in the arguments of p must also be in the arguments of some p_i. If $h = 0$, then each a_i must be a constant, in which case $p(a_1, ..., a_k)$ is called a *fact*. For the rest of the paper, "rule" refers only to the case where $h \geqslant 1$, in which case each $p_i(a_{i1}, ..., a_{ik_i})$ is called a *hypothesis*, and $p(a_1, ..., a_k)$ is called the *conclusion*. For rules with the same hypotheses but different conclusions, we use the shorthand of writing one rule with the same hypotheses but with comma-separated conclusions.

The meaning of a set of rules and facts is the set of facts that are given or can be inferred using the rules.

Terminology. An *IDB (intensional database) predicate* is a predicate defined by rules, and an *EDB (extensional database) predicate* is a predicate for which no rules exist, and only facts are given. An *IDB (EDB) hypothesis* is a hypothesis whose predicate is an IDB (EDB) predicate.

For complexity calculation, we use the following notations.

- `#p`: number of facts of predicate `p`, called *size of* `p`.
- `#dom(p.i)`: size of the domain from which the `i`th argument of predicate `p` takes value.
- `#p.i`: number of values actually taken by the *i*th argument of the facts of predicate `p` (given or inferred).
- $\texttt{\#p.i}_1\texttt{,...,i}_n\texttt{/j}_1\texttt{,...,j}_m$: maximum number of combinations of different values actually taken by the $\texttt{i}_1,...,\texttt{i}_n$th arguments of the facts of predicate `p` (given or inferred), given any fixed value for the $\texttt{j}_1,...,\texttt{j}_m$th arguments.

We assume that hash tables, tries or similar data structures are used so that operations involving a single element of a set take $O(1)$ time.

2.1 Bottom-up evaluation and complexity calculation

Bottom-up evaluation starts with given facts, infers new facts from conclusions of rules whose hypotheses match existing facts, and does so repeatedly until all facts are inferred. In this paper, we use the bottom-up evaluation method of (Liu and Stoller 2009). The time complexity incurred by each rule using this method is bound by the *number of firings* of the rule—the number of combinations of facts that make all hypotheses true. The best complexity is the minimum among all possible decompositions of the rule into rules with at most two hypotheses. However, the number of decompositions of a rule is worse than exponential in the number of hypotheses.

In this subsection, we summarize how to compute the optimal time complexity incurred by a rule with at most two hypotheses with the method in (Liu and Stoller

2009). Note that the number of rules and the arities of predicates are considered constants, not affecting the asymptotic analysis. In the next section, we describe a heuristic algorithm for decomposing rules so that each rule has at most two hypotheses and then calculating the optimal complexity for the decomposition.

There are two forms of rules when the rules are limited to two hypotheses. When a rule has one hypothesis, it is of the form: `p(...) ← q(...)`. The number of times this rule can fire is the number of facts of `q`, therefore the time complexity incurred by this rule is $O(\#\mathtt{q})$. In fact, we can omit the complexity of such rules, because (i) if `q` is an EDB predicate, then all of its facts need to be read in, therefore $O(\#\mathtt{q})$ cost is already incurred by the reading of the input, (ii) if `q` is an IDB predicate, then its size would be bound by the complexity of the rules that infer its facts, and therefore that complexity would already have been included by the complexity of the rules inferring its facts.

When a rule has two hypotheses, it is of the form: $\mathtt{p(...)} \leftarrow \mathtt{q(x_1,...,x_n)}$, $\mathtt{r(y_1,...,y_m)}$. To calculate the number of firings, we can first think of processing the facts of `q` and matching them with facts of the second hypothesis such that the common variables in the hypotheses take the same value. Therefore, only the variables of the second hypothesis not in the first can take values for each fact of `q`. We use C_{12} to denote the set of integers j in $[1,..,m]$ such that $\mathtt{y}_j$ is a variable common to both hypotheses, then the complexity is bounded by $O(\#\mathtt{q}{\times}\#\mathtt{r}.i \notin C_{12}/j \in C_{12})$. Analogously, we can think of processing the facts of `r` and matching them with facts of the first hypothesis. Defining C_{21} analogously to be the set of integers j in $[1,..,n]$ such that $\mathtt{x}_j$ is a common variable, the complexity is also bounded by $O(\#\mathtt{r}{\times}\#\mathtt{q}.i \notin C_{21}/j \in C_{21})$. Since both bounds are upper bounds on the number of firings, then the complexity is bounded by the minimum of the two: $O(\mathtt{min}(\#\mathtt{q} \times \#\mathtt{r}.i \notin C_{12}/j \in C_{12},\ \#\mathtt{r} \times \#\mathtt{q}.i \notin C_{21}/j \in C_{21}))$. For example, for the rule `p(x,z) ← q(x,y), r(y,z).`, the complexity is $O(\mathtt{min}(\#\mathtt{q} \times \#\mathtt{r.2/1},\ \#\mathtt{r} \times \#\mathtt{q.1/2}))$.

3 Handling many hypotheses—applied to pointer analyses for different language features

In general, rules may have many hypotheses. In this section, we describe an algorithm for decomposing rules so that the resulting set of rules has at most two hypotheses and achieves the best complexity among all possible decompositions. We then apply our algorithm coupled with the complexity calculation of bottom-up evaluation to calculate the complexity of may-point-to analysis for object-oriented languages. The algorithm applies also to analyses of other advanced language features, including procedures, arrays, and exceptions as shown in Appendix A of (Tekle and Liu 2016).

3.1 An algorithm for decomposing rules with many hypotheses

Given a set of rules where some rules have more than two hypotheses, each such rule can be *decomposed* so that a set of rules with the same meaning is produced where each rule has two hypotheses. To decompose a rule R, we (1) select two hypotheses h_1 and h_2 of R; (2) create a new intermediate rule R' whose hypotheses are h_1 and

h_2, and whose conclusion c is a new, intermediate predicate whose arguments are the variables occurring in h_1 or h_2 that are also elsewhere in R; (3) replace h_1 and h_2 in R with c; and (4) repeat steps (1)–(3) until R has only two hypotheses.

How two hypotheses are selected at step (1) of each iteration so that the running time of the resulting set of rules is minimized is analogous to the join-order optimization problem on relational database queries, which is well studied with many heuristic algorithms (Selinger *et al.* 1979; Steinbrunn *et al.* 1997). However, most heuristics assume that the sizes of predicates are known in advance, since they consider sizes of only EDB predicates. We propose a new heuristic algorithm that is deterministic and well-suited to Datalog applications. Our algorithm is presented below, where for each substep, we give the rationale.

- If there is any pair of hypotheses such that the variables of one hypothesis is a subset of the variables of the other, we select any one such pair. This ensures that the intermediate rule has no added asymptotic complexity.
- Otherwise, apply the following steps in order, each step applied to the set of pairs selected so far, starting with the set of all pairs, until a unique pair is selected:
  - (a.i) In a rule, a variable is called *removable* for two hypotheses if the variable only appears in those two hypotheses and nowhere else. We select all pairs of hypotheses with the maximum number of removable variables.
  - (a.ii) If domain size information is available, we multiply the sizes of the domains of each removable variable for each pair of hypotheses, and select all pairs with the maximum product. Steps (a.i) and (a.ii) help to minimize the matching of the different values of the removable variables with other hypotheses in the rest of the rule.
  - (b) We select all pairs of hypotheses that contain the maximum number of shared variables between the hypotheses in the pair. This helps to minimize the number of facts iterated over during the evaluation of the intermediate rule to be created.
  - (c.i) We select all pairs of hypotheses that contain the maximum number of EDB hypotheses. This helps to best understand the complexity of the intermediate rule because sizes of EDB predicates are input parameters.
  - (c.ii) We select all pairs of hypotheses in which the product of the sizes of the EDB hypotheses in the pair is the minimum. This helps to minimize the cost of the intermediate rule.
  - (d) As a last resort, we select the pair of the leftmost two hypotheses.

Next, we show applications of this algorithm coupled with evaluation and complexity calculation.

3.2 Andersen's pointer analysis for object-oriented languages and its complexity

Pointer analysis comes in many flavors depending on what it takes into account. An *intraprocedural* analysis only considers a single procedure. An *interprocedural*

```
v_pt(v,h) ← alloc(v,h,m).                             (R1)
v_pt(v,h) ← move(v,v2), v_pt(v2,h).                   (R2)
v_pt(v,h) ← load(v,v2,f), v_pt(v2,h2), f_pt(h2,f,h).  (R3)
f_pt(h,f,h2) ← store(v,f,v2), v_pt(v,h), v_pt(v2,h2). (R4)

int1(v,f,h2) ← load(v,v2,f), v_pt(v2,h2).  (R3/1)  // two decomposed rules for (R3)
v_pt(v,h) ← int1(v,f,h2), f_pt(h2,f,h).    (R3/2)  //
int2(f,v2,h) ← store(v,f,v2), v_pt(v,h).   (R4/1)  // two decomposed rules for (R4)
f_pt(h,f,h2) ← int2(f,v2,h), v_pt(v2,h2).  (R4/2)  //
```

Fig. 1. An intraprocedural may-point-to analysis in Datalog, (R1)–(R4), and decomposed rules.

analysis considers multiple procedures and interactions among them. A *flow-insensitive* analysis does not take control flows into account, whereas a *flow-sensitive* analysis does and produces a may-point-to or must-point-to set for each variable at each program point. A *context-sensitive* analysis takes calling contexts into account, and produces a may-point-to or must-point-to set for each variable for each possible calling context. A *context-insensitive* analysis does not.

For large programs, it is generally understood that flow-sensitivity and context-sensitivity are not feasible. The most well-known flow- and context-insensitive pointer analysis was developed by Andersen (Andersen 1994), and it is considered to offer a sweet spot between precision and efficiency (Hardekopf and Lin 2007). Andersen formulates a may-point-to analysis in terms of type theory, and the formulation corresponds directly to a logical specification in Datalog. This subsection considers an intraprocedural pointer analysis in Datalog for an object-oriented language based on Andersen's analysis as described in (Smaragdakis and Balatsouras 2015). We show the decomposition of the rules and calculate the precise time complexity.

Predicates and rules for Andersen's analysis for OO languages. Each statement has a corresponding fact, shown below. For the first statement, `h`, called a *heap abstraction*, is a new constant created as an abstraction for the set of possible heap objects created by `new` when executing the statement, and `m`, not used in this intraprocedural analysis but used in later analyses, is the method containing the statement:

`v = new Obj()`	`alloc(v,h,m)`
`v = v2`	`move(v,v2)`
`v.f = v2`	`store(v,f,v2)`
`v = v2.f`	`load(v,v2,f)`

For example, method `foo` in Section 1 has three facts: `alloc(o1,h1,foo)`, `alloc(o2,h2,foo)`, and `move(o2,o1)`, where `h1` and `h2` are fresh constants.

The analysis defines the following two predicates and infers facts of them using the Datalog rules `(R1)`–`(R4)` in Fig. 1; additional explanations can be found in (Smaragdakis and Balatsouras 2015).

- `v_pt(v,h)`: variable `v` may point to heap abstraction `h`
- `f_pt(h1,f,h2)`: heap abstraction `h1` may have its field `f` pointing to heap abstraction `h2`

Table 1. *Complexities for an intraprocedural may-point-to analysis*

(R1)	O(#alloc)
(R2)	O(min(#move $\times$ #v_pt.2/1, #v_pt $\times$ #move.1/2))
(R3/1)	O(min(#load $\times$ #v_pt.2/1, #v_pt $\times$ #load.1,3/2))
(R3/2)	O(min(#int1 $\times$ #f_pt.3/1,2, #f_pt $\times$ #int1.1/2,3))
(R4/1)	O(min(#store $\times$ #v_pt.2/1, #v_pt $\times$ #store.2,3/1))
(R4/2)	O(min(#int2 $\times$ #v_pt.2/1, #v_pt $\times$ #int2.1,3/2))

Decomposition of rules and complexity analysis. Using our algorithm for rule decomposition in Section 3.1, we decompose rules (R3) and (R4) as shown in Fig. 1. For our rule decompositions, we note that in the input programs, there are more program points than variables, and more variables than heap abstractions. For (R3), by algorithm step (a.ii), we select the first two hypotheses because v2 is the removable variable whose domain size is maximum. For (R4), we can select the first two hypotheses or the first and the third because v and v2 are removable and their domains are the same, so we select the leftmost two hypotheses that remove v.

We calculate precise complexities for the decomposed rules, and show the results in Table 1. For the rest of the paper, p is the number of program points, v is the number of variables, h is the number of heap abstractions (i.e., #alloc), and f is the number of fields.

The sizes of IDB predicates are bounded by both the complexities of the rules inferring their facts, and the product of the sizes of the domains of their arguments. Sizes #int1 and #int2 are bounded by the complexities of (R3/1) and (R4/1) respectively, because each firing produces at most one new fact; they are also bounded by $O(v \times h \times f)$ based on the domains of their arguments. In all cases, #v_pt is bounded by $O(v \times h)$, and #f_pt is bounded by $O(h^2 \times f)$. These complexities can be factored in when calculating the overall complexities.

Now, we give some insight into the calculated complexities. If we consider the worst case when all predicates are maximized (i.e., they have facts for all possible combinations of their argument values), the complexity of this analysis would be $O(p \times h^2)$. Under various conditions, we can obtain better complexities. For example, if all variables point to a constant number of heap abstractions, i.e., O(#v_pt.2/1)=$O(1)$, then the complexity would be $O(p \times h)$. If, in addition, all fields of variables point to a constant number of heap abstractions, i.e., O(#f_pt.3/1,2)=$O(1)$, then the complexity would be linear in the program size, $O(p)$.

4 Handling rules with function symbols—applied to context-sensitive may-point-to analyses

A context-sensitive may-point-to analysis separates may-point-to sets for executions that map to different contexts, thereby increasing precision. One can consider different types of contexts such as call sites (Shivers 1991; Sharir and Pnueli 1981), objects (Milanova *et al.* 2005), and types (Smaragdakis *et al.* 2011). A rule-based model of context-sensitive analysis is presented in (Smaragdakis and Balatsouras

2015), but the rules contain function symbols, invalidating the polynomial-time evaluation guarantee for pure Datalog; different restrictions to the function symbols are provided to ensure polynomial-time evaluation. In this section, we consider the restriction for the most sophisticated analysis, show how to extend our evaluation to handle function symbols, and calculate the complexities.

Evaluation of Datalog with bounded-size terms. We extend Datalog so that arguments of predicates may be terms, where a *term* is either a constant, a variable, or a function symbol with arguments that are terms. We denote function symbols with uppercase letters[1], and require that each function symbol `F` be of fixed arity, so all occurrences of `F` take the same number of arguments. If a term is a function symbol `F` with arguments, we call it a *term of* `F`. The number of constants and function symbols in a term is called its *size*. The introduction of function symbols to Datalog rules makes the language Turing-complete, therefore invalidating complexity and termination guarantees (Schreye and Decorte 1994). We introduce a sufficient condition for detection of termination in the presence of function symbols, and discuss the evaluation and complexity when termination is guaranteed.

We say that a rule is *size-bounding* for `F` if the sizes of the terms of `F` in the conclusion are guaranteed to be no larger than the size of the term of `F` with the maximum size in the hypotheses. If a set of Datalog rules extended with function symbols is size-bounding for every function symbol, then bottom-up evaluation is guaranteed to terminate. Note that rules with no function symbols in the conclusion are size-bounding by definition.

Given a set of size-bounding Datalog rules, we perform bottom-up evaluation and calculate its complexity exactly as described before. However, the sizes of predicates and domains of predicate arguments need to be made more precise for calculating the number of firings, because they can take on terms as values. For a function symbol `F`, we define `count(F)` to be the number of different terms of `F` that can appear during evaluation. If a size-bounding rule r has a term of `F` in the conclusion distinct from terms of F that appear in its hypotheses, then the contribution of r to `count(F)` is bounded by the product of the domains of the variables that appear in the term of `F` in the conclusion of r. Therefore, `count(F)` is bounded by the sum of such contributions in every rule. The size of the domain of the ith argument of a predicate `p` (`#dom(p.i)`) is bounded by `count(F)`, if there is a rule whose conclusion's predicate is `p` and (i) its ith argument is a term of `F`, or (ii) there is a hypothesis of predicate `q` whose jth argument $\mathtt{a}_j$ is bounded by `count(F)` and the ith argument of the conclusion is $\mathtt{a}_j$.

2-call-site sensitive analysis with a 1-call-site sensitive heap. A pointer analysis is said to be n-call-site sensitive if it tracks the last n method calls leading to the execution of a statement, with an m-call-site sensitive heap if it tracks the last m method calls leading to the creation of a heap object. Out of three context-sensitive analyses in (Smaragdakis and Balatsouras 2015), we consider the most complex one, a 2-call-site sensitive analysis with a 1-call-site sensitive heap. The following

[1] In logic programming, the converse is true; we use this notation to emphasize their presence.

additional kinds of facts are used:

`vcall(v,s,p,m)`	virtual call `v.s(...)` is at program point `p` in method `m`
`htype(h,t)`	heap abstraction `h` has type `t`
`lookup(t,s,m)`	method with signature `s` of type `t` is `m`
`this(m,t)`	`this` variable for method `m` is `t`
`farg(m,n,a)`	method `m`'s `n`th formal argument is `a`
`aarg(p,n,a)`	program point `p` is a call whose `n`th actual argument is `a`
`fret(m,v)`	method `m`'s formal return variable is `v`
`aret(p,v)`	program point `p` is a call that assigns to actual return variable `v`
`astore(v1,v2)`	store into array element as in `v1[..] = v2`
`aload(v1,v2)`	load from array element as in `v1 = v2[..]`
`etype(t,et)`	array type `t` has element type `et`
`stype(t1,t2)`	type `t1` is a subtype of type `t2`
`throw(p,v)`	program point `p` throws variable `v`
`catch(t,p,v)`	for exceptions at program point `p` with arg type `t`, assign arg to `v`
`in(p,m)`	program point `p` is in method `m`

The analysis defines the following predicates and infers facts of them using the rules in Fig. 2.

- `v_pt(v,c,h,hc)`: variable `v` in context `c` may point to heap abstraction `h` in heap context `hc`
- `f_pt(h1,hc1,f,h2,hc2)`: heap abstraction `h1` in heap context `hc1` may have its field `f` pointing to heap abstraction `h2` in heap context `hc2`
- `r(m,c)`: method `m` is reached in context `c`
- `call(p,c1,m,c2)`: program point `p` in context `c1` calls method `m` in context `c2`
- `assign(v1,c1,v2,c2)`: variable `v1` in context `c1` is assigned the value of `v2` in context `c2`

Each rule gives a direct implication based on the meaning of the predicate. For example, rule `(R15)` says: if method `m` is reached in context `P(a,b)`, and variable `v` is assigned a new heap abstraction `h` in method `m`, then `v` in context `P(a,b)` may point to `h` in heap context `a`. Note that a context is a pair represented with function symbol `P` since the analysis is 2-call-site sensitive, and that an initial fact `r(main,P(null,null))` can be used to indicate that method `main` is reached in an initial context where the last two calls before calling `main` are `null`.

Following the method above, we first show that the rules are size-bounding for `P` (the only function symbol). `(R16)`, `(R17)`, `(R18)`, `(R20)`, `(R21)` are trivially size-bounding since they have no function symbols in the conclusion. `(R15)` is size-bounding since the term of `P` appearing in the conclusion is identical to the one in the hypotheses. `(R19)` is size-bounding since (i) `P(a,b)` in the conclusion is identical to an occurrence of `P` in the hypotheses, and (ii) for `P(p,a)`, `p` is a program point (i.e., a constant), and therefore cannot have a larger size than `b`, and the size of `P(p,a)` is no more than `P(a,b)`.

Next, we determine which arguments of which predicates are bounded by `count(P)`. These are `v_pt.2` due to `(R15)` and `(R19)`, `r.2` due to `(R19)`, `call.2` and `call.4` due to `(R19)`, and `assign.2` and `assign.4` due to `(R20)` and `(R21)` .

```
v_pt(v,P(a,b),h,a) ← r(m,P(a,b)), alloc(v,h,m).                              (R15)
v_pt(v,c,h,hc) ← move(v,v2), v_pt(v2,c,h,hc).                                 (R16)
f_pt(h1,hc1,f,h2,hc2) ← store(v1,f,v2), v_pt(v2,c,h2,hc2), v_pt(v1,c,h1,hc1). (R17)
v_pt(v,c,h,hc) ← load(v,v2,f), v_pt(v2,c,h2,hc2), f_pt(h2,hc2,f,h,hc).        (R18)
r(m,P(p,a)), v_pt(t,P(p,a),h,hc), call(p,P(a,b),m2,P(p,a)) ← vcall(v,s,p,m1),
     r(m1,P(a,b)), v_pt(v,P(a,b),h,hc), htype(h,ht), lookup(ht,s,m2), this(m2,t).(R19)
assign(v1,c2,v2,c1) ← call(p,c1,m,c2), farg(m,n,v1), aarg(p,n,v2).           (R20)
assign(v1,c1,v2,c2) ← call(p,c1,m,c2), aret(p,v1), fret(m,v2).               (R21)
v_pt(v,c,h,hc) ← assign(v,c,v2,c2), v_pt(v2,c2,h,hc).                         (R22)

// decomposed rules:
int23(v1,v2,h1,hc1,h2,hc2) ← v_pt(v2,c,h2,hc2), v_pt(v1,c,h1,hc1).      (R17/1)
f_pt(h1,hc1,f,h2,hc2) ← store(v1,f,v2), int23(v1,v2,h1,hc1,h2,hc2).     (R17/2)
int24(v,f,c,h2,hc2) ← load(v,v2,f), v_pt(v2,c,h2,hc2).                  (R18/1)
v_pt(v,c,h,hc) ← int24(v,f,c,h2,hc2), f_pt(h2,hc2,f,h,hc).              (R18/2)
int25(s,p,m1,P(a,b),h,hc) ← vcall(v,s,p,m1), v_pt(v,P(a,b),h,hc).       (R19/1)
int26(s,p,P(a,b),h,hc) ← int25(s,p,m1,P(a,b),h,hc), r(m1,P(a,b)).       (R19/2)
int27(s,p,P(a,b),h,hc,ht,m2) ← int26(s,p,P(a,b),h,hc), lookup(ht,s,m2). (R19/3)
int28(s,p,P(a,b),h,hc,m2) ← int27(s,p,P(a,b),h,hc,ht,m2), htype(h,ht).  (R19/4)
r(m,P(p,a)), v_pt(t,P(p,a),h,hc), call(p,P(a,b),m2,P(p,a)) ←
                            int28(s,p,P(a,b),h,hc,m2), this(m2,t).      (R19/5)
int29(c1,m,c2,n,v2) ← call(p,c1,m,c2), aarg(p,n,v2).                    (R20/1)
assign(v1,c2,v2,c1) ← int29(c1,m,c2,n,v2), farg(m,n,v1).                (R20/2)
int30(c1,m,c2,v1) ← call(p,c1,m,c2), aret(p,v1).                        (R21/1)
assign(v1,c1,v2,c2) ← int30(c1,m,c2,v1), fret(m,v2).                    (R21/2)
```

Fig. 2. A 2-call-site sensitive analysis with a 1-call-site sensitive heap, and decomposed rules.

Finally, we determine `count(P)`. The only rule whose conclusion contains a term of `P` distinct from the terms in its hypotheses is `(R19)`. The variables of this term of `P` are `p` and `a`. The source of `p` is the third argument of the first hypothesis, therefore its domain size is `#vcall.3`. The source of `a` is the first argument in the terms of `P` in the second and third hypotheses, but this argument of `P` is, as just analyzed, only from the third argument of the first hypothesis. Therefore, `count(P)` is $O((\texttt{\#vcall.3})^2)$.

We decompose the rules with our algorithm as shown in Fig. 2, and the calculated complexities are shown in Table 2.

5 Handling rules with universal quantification—applied to flow-sensitive must-point-to analysis

Must-point-to analysis determines the heap abstractions that a pointer variable or expression must refer to, as opposed to may refer to, in all program executions. Flow-sensitive analysis determines analysis results specific to each program point, as opposed to one global result for the program. Therefore, flow-sensitive must-point-to analysis can give significantly more certain results that complement flow-insensitive may-point-to alias analysis. This analysis poses two new challenges:

1. The analysis is much more complex, requiring extensions to Datalog with universal quantification and negation.
2. The analysis algorithm is much more sophisticated, requiring new techniques to keep the complexity from increasing.

Table 2. *Complexities for a 2-call-site sensitive analysis with a 1-call-site sensitive heap*

(R15)	O(min(#r × #alloc.1,2/3, #alloc × #r.2/1))
(R16)	O(min(#move × #v_pt.2,3,4/1, #v_pt × #move.1/2))
(R17/1)	O(#v_pt×#v_pt.1,3,4/2)
(R17/2)	O(min(#store × #int23.3,4,5,6/1,2, #int23 × #store.2/1,3))
(R18/1)	O(min(#load × #v_pt.2,3,4/1, #v_pt × #load.1,3/2))
(R18/2)	O(min(#int24 × #f_pt.4,5/1,2,3, #f_pt × #int24.1,3/2,4,5))
(R19/1)	O(min(#vcall × #v_pt.2,3,4/1, #v_pt × #vcall.2,3,4/1))
(R19/2)	O(#int25)
(R19/3)	O(min(#int26 × #lookup.1,3/2, #lookup × #int26.2,3,4,5/1))
(R19/4)	O(#int27)
(R19/5)	O(min(#int28 × #this.2/1, #this × #int28.1,2,3,4,5/6))
(R20/1)	O(min(#call × #aarg.2,3/1, #aarg × #call.2,3,4/1))
(R20/2)	O(min(#int29 × #farg.3/1,2, #farg × #int29.1,3,5/2,4))
(R21/1)	O(min(#call × #aret.2/1, #aret × #call.2,3,4/1))
(R21/2)	O(min(#int30 × #fret.2/1, #fret × #int30.1,3,4/2))

```
must_pt(var,h) ← alloc(_,var,h,_).
must_pt(to,h) ← move(_,to,from), must_pt(from,h).
must_pt(to,h) ← phi(_,to,from1,from2), must_pt(from1,h), must_pt(from2,h).
must_pt(to,h2) ← load(i,to,v,f), must_pt(v,h), f_must_pt(i,h,f,h2).
f_must_pt(i,h,f,h2) ← store(i,v,f,from), must_pt(from,h2), must_pt(v,h).
f_must_pt(j,h,f,h2) ←
     next(_,j), f_must_pt(_,h,f,h2), (forall i: next(i,j) → f_must_pt(i,h,f,h2)),
     not store(j,_,f,_), not vcall(_,_,j,_), not alloc(j,_,h), not alloc(j,_,h2).
f_must_pt(j,h,f,h2) ←
     next(_,j), f_must_pt(_,h,f,h2), (forall i: next(i,j) → f_must_pt(i,h,f,h2)),
     store(j,v,f,_), must_pt(v,h3), h3 != h.
```

Fig. 3. A flow-sensitive must-point-to analysis in Datalog with universal quantification, negation, and inequality.

Specification using Datalog rules with universal quantification and negation. The analysis is specified using seven rules (Smaragdakis and Balatsouras 2015), shown in Fig. 3 (after the changes noted in the third paragraph below). The last two rules are the core of the analysis. The first five rules define `must_pt` and a simple case of `f_must_pt`, where `alloc`, `move`, `load`, and `store` are as in Section 3.2 except with an additional first argument indicating the program point, and `phi` is an instruction for merging the values of two variables. The first five rules are simple Datalog rules with one, two, or three hypotheses; they can be analyzed using the method in Sections 2.1 and 3.1, yielding a time complexity of $O(p \times h^2)$. This section focuses on the two core rules, which are Datalog extended with universal quantification, simple negation, as well as inequality. These two rules are the core of the flow-sensitive analysis because they infer field-must-point-to information for each program point by combining information from all its predecessor points.

The first core rule says that, just after instruction `j`, `h` must point via its field `f` to `h2` if (1) `j` is the next instruction of some instruction, (2) `h` must point via `f` to `h2` just after some instruction, (3) for all instructions `i` just before `j`, `h` must point via `f` to `h2` at `i`, and (4) `j` is not a `store`, `vcall`, or `alloc` instruction that can change the must-point-to information.

The second core rule concludes the same if the same conditions hold except that `j` is a `store` instruction into field `f` of `v`, and `v` must point to a heap abstraction `h3` that is not `h`.

Note that, compared to the original two core rules (Smaragdakis and Balatsouras 2015), we added the first two hypotheses in each rule and moved the conditions about `j` out of the universal quantification. The two new hypotheses bind the free variables not bound by the universal quantifications, and are necessary for the rules to be correct; without them, the universal quantification returns true when no `i` satisfies its domain condition `next(i,j)`, which would lead to `f_must_pt` to hold for all values of `h`, `f`, and `h2` for any `j` for which no `i` satisfies `next(i,j)`. The conditions about `j` are moved out because they do not depend on the universally quantified variable `i`. These also show that the analysis is complex and universal quantification is challenging.

Analysis algorithm for universal quantification and inequality. Despite negation and inequality in the core rules, the set of `f_must_pt` facts that can be inferred still increases monotonically. Therefore, the set can be computed as a least fixed point as for Datalog. However, if computed straightforwardly, universal quantification adds a linear factor after each `f_must_pt` fact is added. We show how to compute it, as well as the negation and inequality, incrementally in $O(1)$ time.

Consider the universal quantification, in both core rules:

```
(forall i: next(i,j) → f_must_pt(i,h,f,h2))
```

To compute it efficiently, we maintain the following four auxiliary invariants:

```
prev[j] = {i: (i,j) in next}  and
prev_count[j] = #prev[j],  for j in next.2

prev_pt[j,h,f,h2] = {i: (i,j) in next, (i,h,f,h2) in f_must_pt}  and
prev_pt_count[j,h,f,h2] = #prev_pt[j,h,f,h2],  for j in next.2, (h,f,h2)
   in f_must_pt.2,3,4
```

and replace the universal quantification with the following $O(1)$ time test between two aggregate count values:

```
prev_count[j] = prev_pt_count[j,h,f,h2]
```

Variables `prev` and `prev_count` for the first two invariants are initialized by iterating over each element `(i,j)` of input `next`, adding `i` to `prev[j]` and incrementing `prev_count[j]`, in a total of $O($`#next`$)$ time. The next two invariants are maintained incrementally at addition of `(i,h,f,h2)` to `f_must_pt` as follows, taking a total of $O($`#next.2/1` $\times$ `f_must_pt`$)$ time overall all additions:

```
for j in next[i]:                   // use next to get each next node
   prev_pt[j,h,f,h2] ∪= {i}         // i is new to h,f,h2 because (i,h,f,h2) is new
   prev_pt_count[j,h,f,h2] += 1     // increment the corresponding count by 1
```

The first core rule now becomes Datalog with simple negations as $O(1)$ time tests, and with the universal quantification as an $O(1)$ time equality test between two counts. Its total time complexity is $O($`#next.2` $\times$ `#f_must_pt.2,3,4`$)$.

The second core rule is similar in terms of the universal quantification, but it does not have simple negations but an inequality. We handle the inequality specially, replacing the last two hypotheses on the last line with the following, removing the extra variable `h3`:

```
must_pt[v] - {h} != {} //use must_pt[v] to get heap abstractions that v must point to
```

There is only one value for `v` in a `store` instruction, and the above element subtraction and test take $O(1)$ time. So the total time complexity of this rule is again $O(\texttt{\#next.2} \times \texttt{\#f\_must\_pt.2,3,4})$.

Complexity guarantees. Summing all time complexities together, from initialization, maintaining auxiliary invariants, and using the two resulting rules, yields the total time complexity

$$O(\texttt{\#next} + \texttt{\#next.2/1} \times \texttt{\#f\_must\_pt} + \texttt{\#next.2} \times \texttt{\#f\_must\_pt.2,3,4})$$

`#next` is bounded by $O(p)$, the size of the program. `f_must_pt.2,3,4` is bounded by the domain sizes of its three arguments $O(h \times f \times h)$. Therefore, the total time complexity is $O(p \times h^2 \times f)$.

6 Additional pointer analyses and summary of complexity analysis results

Besides the 3 pointer analyses discussed, we also studied the remaining 6 analyses in (Smaragdakis and Balatsouras 2015), including 3 in Appendix A of (Tekle and Liu 2016); we do not present the rest in detail because they are simpler and do not illustrate additional logic rule features for complexity analysis. There are also other analyses that can be specified using Datalog rules, such as the context-free-language formulation in (Zheng and Rugina 2008). We believe that the reader can follow our method to produce a set of rules and analyze their complexities easily.

In addition to the precise complexities that we calculated, here we also present the worst-case complexities in simpler terms, and provide conditions under which the complexities are linear or quadratic. Table 3 summarizes, for each analysis, the features used, maximum number of hypotheses in the rules, worst-case complexities, and complexities conditioned on constraints on sizes of predicates. We denote an n-call-site sensitive analysis with an m-call-site sensitive heap, as (n, m)-context. The conditions used are as follows, where the conditions on all the EDB predicates are typical for real programs.

(C1): $O(\texttt{\#v\_pt.2/1}) = O(1)$
(C2): $O(\texttt{\#f\_pt.3/1,2}) = O(1)$
(C3): $O(\texttt{\#lookup.1,3/2}) = O(\texttt{\#this.2/1}) = O(\texttt{\#call.2/1}) = O(\texttt{\#farg.3/1,2}) = O(\texttt{\#fret.2/1}) = O(1)$
(C4): $O(\texttt{\#htype.2/1}) = O(\texttt{\#in.2/1}) = O(\texttt{\#t\_pt.2/1}) = O(\texttt{\#throw.2/1}) = O(1)$
(C5): $O(\texttt{\#must\_pt.2/1}) = O(1)$
(C6): $O(\texttt{\#f\_must\_pt.2,3,4}) = O(h)$

(C1) says that each variable may point to a constant number of heap abstractions. (C2) says that each field of each variable may point to a constant number of heap abstractions. The rest are similar.

Table 3. *Summary of complexities. Recall p, h, and f are number of program points, heap abstractions, and fields, respectively*

Analysis	Features	#Hypo.	Worst-case	Conditional
Andersen OO (Sec. 3)	Pure Datalog	3	$O(p \times h^2)$	$O(p \times h)$ if (C1), $O(p)$ if (C1) and (C2)
Interprocedural (Sec. A.1)[2]	Pure Datalog	6	$O(p^2 \times h)$	$O(p)$ if (C1), (C2) and (C3)
Arrays (Sec. A.2)[2]	Pure Datalog	7	$O(p^2 \times h)$	$O(p)$ if (C1), (C2) and (C3)
Exceptions (Sec. A.3)[2]	Negation	5	$O(p^2 \times h)$	$O(p)$ if (C1) and (C3)
Reflection	Pure Datalog	9	$O(p^3 \times h)$	$O(p \times h)$ obtainable[3]
(0,1)-context	Function symbols	6	$O(p^3 \times h^2)$	–
(1,1)-context	Function symbols	6	$O(p^4 \times h^2)$	–
(2,1)-context (Sec. 4)	Function symbols	6	$O(p^5 \times h^2)$	–
Flow must (Sec. 5)	Univ. quant., negation, inequality	8	$O(p \times h^2 \times f)$	$O(p \times h)$ if (C5) and (C6)

Complexities of pointer analysis with constraints on the sizes of program parameters have been studied. One can obtain the complexities achieved in such studies using our method, and substituting the relevant complexity parameters in our analysis with the constraints. In (Sridharan and Fink 2009), the authors present an algorithm for Andersen's analysis, which runs in $O((v+h)^2)$ time for k-sparse programs, where v is the number of variables and h is the number of heap abstractions. The definition of k-sparse programs has two constraints: For our rules in Figure 1, the first constraint implies $O(\texttt{\#store.2,3/1}) = O(\texttt{\#load.1,3/2}) = O(1)$, and the second constraint implies that $\texttt{\#move}+\texttt{\#int1}+\texttt{\#int2} \leqslant O(v+h)$. Substituting these constraints in our complexity analysis in Table 1, we obtain $O((v+h) \times (\texttt{\#v\_pt.2/1} + \texttt{\#f\_pt.3/1,2}))$, which is in the worst case $O((v+h) \times h)$. Thus, we obtain a better and more precise complexity than (Sridharan and Fink 2009).

For Andersen OO analysis, interprocedural analysis, and (0,1)-context-sensitive analysis, the worst-case complexities when parametrized by only program size, n, are known—an upper bound of $O(n^3)$ for the first two (Andersen 1994) and $O(n^5)$ for the third (Wilson and Lam 1995). However, to our knowledge, we present more precise complexities for these analyses for the first time, and for the other analyses, we present complexities for the first time. Our complexity results are improvements since they are tighter than known worst-case complexities, and when

[2] In Appendix A of (Tekle and Liu 2016).
[3] The conditions to obtain this complexity involve EDB predicates not discussed in this paper.

our fine-grained analyses are used, the running time of the analyses can be better understood.

7 Related work and conclusion

We discuss related work on Datalog evaluation, applications of rules to pointer analysis, precise complexities for pointer analyses, and directions for future work.

Evaluation of Datalog has been studied for a long time (Ceri *et al.* 1990). Optimal bottom-up evaluation of Datalog rules with complexity guarantees is first given in (Liu and Stoller 2009), but no algorithm is given for decomposing rules except for trying all decompositions. We build on this method for evaluating Datalog rules and calculating complexities, but extend it to handle rules with many hypotheses and other Datalog extensions. Our new algorithm and method are able to obtain new or more precise complexities compared with the best previous complexities, as discussed in Section 6.

Formulation of various static program analyses as rules has been studied. In particular, Andersen's pointer analysis (Andersen 1994) was formulated as deductive rules in (Heintze and Tardieu 2001), and given as logic rules in (Saha and Ramakrishnan 2005). Andersen's analysis with many flavors was given in a recent survey (Smaragdakis and Balatsouras 2015), on which we base our study. The fact that the time complexity of Andersen's analysis is worst-case cubic has been known since the original introduction. (Sridharan and Fink 2009) notes that the typical behavior is different from the worst case and proves that under certain conditions the analysis is quadratic. We give the precise time complexities for Andersen's analysis for an object-oriented language, and show also precise conditions, additional to existing literature, under which the complexities are linear or quadratic directly as results of our complexity analyses. We also obtain precise complexities for pointer analyses for additional language features, and provide methods for handling extensions to Datalog such as function symbols when such extensions are necessary to implement the analyses.

Must-point-to analyses are more complex, and rules modeling the analyses involve universal quantification and inequality beyond pure Datalog. (Hind *et al.* 1999) gives an $O(n^5)$ algorithm for a must-alias analysis (closely related to must-point-to analyses); using methods that are also employed in optimal bottom-up evaluation of Datalog, (Goyal 2005) improves this complexity to $O(n^3)$. In this paper, we show how to handle must-point-to analysis expressed using rules extended with universal quantification and a special inequality, and provide precise complexity analyses for our efficient implementation. Transforming quantifications into aggregate queries such as counts has been used in other applications, e.g., distributed algorithms (Liu *et al.* 2012), but how to handle inequality in general is a subject for future study.

Future directions include analyzing and optimizing the space complexity of pointer analyses, especially to remove unnecessary intermediate predicates introduced for rules with many hypotheses, and optimization of demand-driven pointer analysis via queries, e.g., by using the methods of (Tekle and Liu 2010) and (Tekle and Liu 2011).

References

Andersen, L. O. 1994. Program analysis and specialization for the c programming language. Ph.D. thesis, DIKU, University of Copenhagen, Copenhagen, Denmark.

Avots, D., Dalton, M., Livshits, V. B. and Lam, M. S. 2005. Improving software security with a C pointer analysis. In *Prof. of the 27th Intl. Conf. on Software Engineering*, 332–341.

Ceri, S., Gottlob, G. and Tanca, L. 1990. *Logic Programming and Databases*. Springer.

Diwan, A., McKinley, K. S. and Moss, J. E. B. 1998. Type-based alias analysis. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 106–117.

Ghiya, R. and Hendren, L. J. 1998. Putting pointer analysis to work. In *Proc. of the 25th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, 121–133.

Ghiya, R., Lavery, D. M. and Sehr, D. C. 2001. On the importance of points-to analysis and other memory disambiguation methods for C programs. In *Proc. of the 2001 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 47–58.

Gorbovitski, M., Liu, Y. A., Stoller, S. D., Rothamel, T. and Tekle, T. 2010. Alias analysis for optimization of dynamic languages. In *Proceedings of the 6th Symposium on Dynamic Languages*. ACM Press, 27–42.

Goyal, D. 2005. Transformational derivation of an improved alias analysis algorithm. *Higher-Order and Symbolic Computation 18,* 1–2, 15–49.

Hardekopf, B. and Lin, C. 2007. The ant and the grasshopper: Fast and accurate pointer analysis for millions of lines of code. In *Proc. of the ACM SIGPLAN 2007 Conf. on Programming Language Design and Implementation*, 290–299.

Heintze, N. and Tardieu, O. 2001. 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*, 254–263.

Hind, M. 2001. Pointer analysis: Haven't we solved this problem yet? In *Proc. of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering*, 54–61.

Hind, M., Burke, M. G., Carini, P. R. and Choi, J. 1999. Interprocedural pointer alias analysis. *ACM Trans. Program. Lang. Syst. 21,* 4, 848–894.

Hind, M. and Pioli, A. 2000. Which pointer analysis should I use? In *Proc. of the 2000 ACM SIGSOFT Intl. Symp. on Software Testing and Analysis*, 113–123.

Landi, W. 1992. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems 1,* 4, 323–337.

Liu, Y. A. and Stoller, S. D. 2009. From Datalog rules to efficient programs with time and space guarantees. *ACM Trans. Program. Lang. Syst. 31,* 6 (Aug.), 21:1–21:38.

Liu, Y. A., Stoller, S. D., Lin, B. and Gorbovitski, M. 2012. From clarity to efficiency for distributed algorithms. In *Proc. of the 27th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications*, 395–410.

Milanova, A., Rountev, A. and Ryder, B. G. 2005. Parameterized object sensitivity for points-to analysis for java. *ACM Trans. Softw. Eng. Methodol. 14,* 1, 1–41.

Pearce, D. J., Kelly, P. H. J. and Hankin, C. 2007. Efficient field-sensitive pointer analysis of C. *ACM Trans. Program. Lang. Syst. 30,* 1.

Ramalingam, G. 1994. The undecidability of aliasing. *ACM Trans. Program. Lang. Syst. 16,* 5, 1467–1471.

Sagiv, S., Reps, T. W. and Wilhelm, R. 1998. Solving shape-analysis problems in languages with destructive updating. *ACM Trans. Program. Lang. Syst. 20,* 1, 1–50.

Saha, D. and Ramakrishnan, C. R. 2005. Incremental and demand-driven points-to analysis using logic programming. In *Proc. of the 7th Intl. ACM SIGPLAN Conf. on Principles and Practice of Declarative Programming*, 117–128.

Schreye, D. D. and Decorte, S. 1994. Termination of logic programs: The never-ending story. *The Journal of Logic Programming 19/20*, 199–260.

Selinger, P. G., Astrahan, M. M., Chamberlin, D. D., Lorie, R. A. and Price, T. G. 1979. Access path selection in a relational database management system. In *Proc. of the 1979 ACM SIGMOD Intl. Conf. on Management of Data*, 23–34.

Shapiro, M. and Horwitz, S. 1997. The effects of the precision of pointer analysis. In *Proc. of the 4th Intl. Symp. on Static Analysis*, 16–34.

Sharir, M. and Pnueli, A. 1981. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications.* Prentice-Hall, Chapter 7, 189–233.

Shivers, O. G. 1991. Control-flow analysis of higher-order languages of taming lambda. Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA, USA.

Smaragdakis, Y. and Balatsouras, G. 2015. Pointer analysis. *Foundations and Trends in Programming Languages 2,* 1, 1–69.

Smaragdakis, Y., Bravenboer, M. and Lhoták, O. 2011. Pick your contexts well: Understanding object-sensitivity. In *Proc. of the 38th Symp. on Principles of Programming Languages*, 17–30.

Sridharan, M. and Fink, S. J. 2009. The complexity of Andersen's analysis in practice. In *Proc. of the 16th Intl. Symp. on Static Analysis*, 205–221.

Steensgaard, B. 1996. Points-to analysis in almost linear time. In *Conf. Record of the 23rd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, 32–41.

Steinbrunn, M., Moerkotte, G. and Kemper, A. 1997. Heuristic and randomized optimization for the join ordering problem. *The VLDB Journal 6,* 3, 191–208.

Tekle, K. T. and Liu, Y. A. 2010. Precise complexity analysis for efficient Datalog queries. In *Proc. of the 12th Intl. ACM SIGPLAN Conf. on Principles and Practice of Declarative Programming*, 35–44.

Tekle, K. T. and Liu, Y. A. 2011. More efficient Datalog queries: Subsumptive tabling beats magic sets. In *Proc. of the 2011 ACM SIGMOD Intl. Conf. on Management of Data*, 661–672.

Tekle, K. T. and Liu, Y. A. 2016. Precise Complexity Guarantees for Pointer Analysis via Datalog with Extensions. *ArXiv e-prints*.

Wilson, R. P. and Lam, M. S. 1995. Efficient context-sensitive pointer analysis for C programs. In *Proc. of the ACM SIGPLAN'95 Conf. on Programming Language Design and Implementation*, 1–12.

Zheng, X. and Rugina, R. 2008. Demand-driven alias analysis for C. In *Proc. of the 35th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, 197–208.