Pick Your Contexts Well: Understanding Object-Sensitivity

The Making of a Precise and Scalable Pointer Analysis

Yannis Smaragdakis
Department of Computer Science,
University of Massachusetts,
Amherst, MA 01003, USA
and Department of Informatics,
University of Athens, 15784, Greece
yannis@cs.umass.edu—smaragd@di.uoa.gr

Martin Bravenboer
LogicBlox Inc.
Two Midtown Plaza
Atlanta, GA 30309, USA
martin.bravenboer@acm.org

Ondřej Lhoták
David R. Cheriton School of
Computer Science
University of Waterloo
Waterloo, ON N2L 3G1, Canada
olhotak@uwaterloo.ca

Abstract

Object-sensitivity has emerged as an excellent context abstraction for points-to analysis in object-oriented languages. Despite its practical success, however, object-sensitivity is poorly understood. For instance, for a context depth of 2 or higher, past scalable implementations deviate significantly from the original definition of an object-sensitive analysis. The reason is that the analysis has many degrees of freedom, relating to which context elements are picked at every method call and object creation. We offer a clean model for the analysis design space, and discuss a formal and informal understanding of object-sensitivity and of how to create good object-sensitive analyses. The results are surprising in their extent. We find that past implementations have made a sub-optimal choice of contexts, to the severe detriment of precision and performance. We define a "full-object-sensitive" analysis that results in significantly higher precision, and often performance, for the exact same context depth. We also introduce "type-sensitivity" as an explicit approximation of object-sensitivity that preserves high context quality at substantially reduced cost. A type-sensitive points-to analysis makes an unconventional use of types as context: the context types are not dynamic types of objects involved in the analysis, but instead upper bounds on the dynamic types of their allocator objects. Our results expose the influence of context choice on the quality of points-to analysis and demonstrate type-sensitivity to be an idea with major impact: It decisively advances the state-of-the-art with a spectrum of analyses that simultaneously enjoy speed (several times faster than an analogous object-sensitive analysis), scalability (comparable to analyses with much less context-sensitivity), and precision (comparable to the best object-sensitive analysis with the same context depth).

Categories and Subject Descriptors F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program Analysis
; D.3.1 [*Programming Languages*]: Formal Definitions and Theory—Semantics

General Terms Algorithms, Languages, Performance

POPL'11, January 26–28, 2011, Austin, Texas, USA.

1. Introduction

Points-to analysis (or *pointer analysis* in our context) is one of the most fundamental static program analyses. Points-to analysis consists of computing a static abstraction of all the data that a pointer expression (or just a variable, without loss of generality) can point to during program run-time. The analysis forms the basis for practically every other program analysis and is closely inter-related with mechanisms such as call-graph construction, since the values of a pointer determine the target of dynamically resolved calls, such as object-oriented dynamically dispatched method calls or functional lambda applications. By nature, the entire challenge of points-to analysis is to pick judicious approximations. Intractability lurks behind any attempt to track program control- or data-flow precisely. Furthermore, the global character and complicated nature of the analysis make it hard to determine how different analysis decisions interact with various language features. For object-oriented and functional languages, *context-sensitivity* is a general approach that achieves tractable and usefully high precision. Context-sensitivity consists of qualifying local program variables, and possibly (heap) object abstractions, with context information: the analysis collapses information (e.g., "what objects this method argument can point to") over all possible executions that result in the same context, while separating all information for different contexts. Two main kinds of context-sensitivity have been explored: *call-site sensitivity* [18, 19] and *object-sensitivity* [13].

Ever since the introduction of object-sensitivity by Milanova et al. [13], there has been accumulating evidence [3, 7, 8, 10, 14] that it is a superior context abstraction for object-oriented programs, yielding high precision relative to cost. The success of object-sensitivity has been such that, in current practice, object-sensitive analyses have almost completely supplanted traditional call-site sensitive/*k*CFA analyses for object-oriented languages. This paper is concerned with understanding object-sensitivity in depth, formalizing it conveniently, and exploring design choices that produce even more scalable and precise analyses than current practice.

What is object-sensitivity at a high level? Perhaps the easiest way to describe the concept is by analogy and contrast to the better-known call-site sensitivity. A call-site sensitive/*k*CFA analysis uses method call-sites (i.e., labels of instructions that may call the method) as context elements. That is, in OO terms, the analysis separates information on local variables (e.g., method arguments) per call-stack (i.e., sequence of k call-sites) of method invocations that led to the current method call. Similarly, the analysis separates information on heap objects per call-stack of method invocations that led to the object's allocation. For instance, in the code example below, a 1-call-site sensitive analysis (unlike a *context-insensitive*

analysis) will distinguish the two call-sites of method `foo` on lines 7 and 9. This means that the analysis will treat `foo` separately for two cases: that of its argument, `o`, pointing to anything `someobj1` may point to, and that of `o` pointing to anything `someobj2` may point to.

```
1  class A {
2   void foo(Object o) { ... }
3  }
4
5  class Client {
6   void bar(A a1, A a2) { ...
7     a1.foo(someobj1);
8     ...
9     a2.foo(someobj2);
10  }
11 }
```

In contrast, object-sensitivity uses object allocation sites (i.e., labels of instructions containing a `new` statement) as context elements. (Hence, a better name for "object-sensitivity" might have been "allocation-site sensitivity".) That is, when a method is called on an object, the analysis separates the inferred facts depending on the allocation site of the receiver object (i.e., the object on which the method is called), as well as other allocation sites used as context. Thus, in the above example, a 1-object-sensitive analysis will analyze `foo` separately depending on the allocation sites of the objects that `a1` and `a2` may point to. It is not apparent from the above fragment neither whether `a1` and `a2` may point to different objects, nor to how many objects: the allocation site of the receiver object may be remote and unrelated to the method call itself. Similarly, it is not possible to compare the precision of an object-sensitive and a call-site sensitive analysis in principle. In this example, it is not even clear whether the object sensitive analysis will examine all calls to `foo` as one case, as two, or as many more, since this depends on the allocation sites of all objects that the analysis itself computes to flow into `a1` and `a2`.

Note that our above description has been vague: "the analysis separates ... facts depending on the allocation site of the receiver object ... as well as other allocation sites". What are these "other allocation sites"? The first contribution of our paper consists of recognizing that there is confusion in the literature regarding this topic. The original definition of object-sensitivity (see [13], Fig.6, p.12.) defines the context of a method call to be the allocation site of the receiver object *obj*, the allocation site of the allocator object (*obj′*) of *obj*, (i.e., the receiver object of the method that made the allocation of *obj*), the allocation site of the allocator object of *obj′*, and so on. Nevertheless, subsequent "object-sensitive" analyses (e.g., [5, 8, 10, 20], among many) maintain the fundamental premise of using allocation sites as context elements, yet differ in which allocation sites are used. For instance, the 2-object-sensitive analysis in the Paddle framework [7, 9] uses as method context the allocation site of the receiver object and the allocation site of the *caller* object (i.e., an object of class `Client`, and not `A`, in our example).

In this paper, we offer a unified formal framework that captures the object-sensitive analyses defined in the literature and allows a deeper understanding of their differences. Additionally, we implement an array of object-sensitive analyses and draw insights about how the choice of context relates to scalability and precision. We discover that the seemingly simple difference of how an analysis context is chosen results in large differences in precision and performance. We use the name *full-object sensitivity* to refer to (a slight generalization of) the original statement of object-sensitivity by Milanova et al. We argue that full-object sensitivity is an excellent choice in the design space, while the choice of context made in past actual implementations is sub-optimal and results in substantial loss of precision. Concretely, a practical outcome of our work is to establish a 2-full-object-sensitive analysis with a 1-object sensitive heap (shortened to "2full+1H") as an analysis that is often (though not always) feasible with current technology and impressively precise.

Perhaps even more importantly, our understanding of the impact of context on the effectiveness of an analysis leads to defining a new variant that combines scalability with good precision. Namely, we introduce the idea of a *type-sensitive* analysis, which is defined to be directly analogous to an object-sensitive analysis, yet approximates (some) context elements using types instead of full allocation sites. In contrast to past uses of types in points-to analysis (e.g., [1, 15, 21] and see Ryder [17] for several examples) we demonstrate that the types used as contexts should *not* be the types of the corresponding objects. Instead, the precision of our type-sensitive analysis is due to replacing the allocation site of an object *o* (which would be used as context in an object-sensitive analysis) with an upper-bound of the dynamic type of *o*'s allocator object. The result is a substantial improvement that establishes a new sweet spot in the practical tradeoff of points-to analysis precision and performance.

In summary, our work makes the following contributions:

- We offer a better understanding of the concept and variations of object-sensitive points-to analyses. Our understanding relies on a precise formalism that captures the different object-sensitive analyses, as well as on informal insights.
- We identify the differences in past object-sensitive analyses and analyze the influence of these differences on precision and performance. We argue that full-object-sensitivity is a substantially better choice than others used in actual practice. We validate the impact of full-object-sensitivity for the case of a context depth of 2. The difference is significant in terms of precision and scalability. Our results help establish a 2full+1H analysis as the state-of-the-art for precision in object-oriented programs, among analyses that are often practically feasible.
- We introduce type-sensitivity, as a purposeful collapsing of the context information of an object-sensitive analysis, in order to improve scalability. We discuss what is a good type-sensitive context and show that the straightforward option (of replacing an object by its type) is catastrophically bad. Instead, we identify an excellent choice of type context and demonstrate that it yields a surprisingly ideal combination of precision and scalability. A type-sensitive analysis for a context depth of 2 is several times (2x to unboundedly high) faster than a corresponding object-sensitive analysis, while keeping almost the same precision. In fact, the run-time performance and scalability of a type-sensitive analysis often exceed those of a cheap object-sensitive analysis of a lower context depth, while yielding vastly more precision.

2. Formalizing Object-Sensitivity and Variations

We formalize analyses using an abstract interpretation over a simplified base language that closely captures the key operations that practical points-to analyses perform. For this, we use Featherweight Java [6] in "A-Normal" form. A-Normal Featherweight Java is identical to ordinary Featherweight Java, except that arguments to a function call must be atomically evaluable. For example, the body `return f.foo(b.bar());` becomes `b1 = b.bar(); f1 = f.foo(b1); return f1;`. This shift does not change the expressive power of the language or the nature of the analysis, but it simplifies the semantics and brings the language closer to the intermediate languages that practical points-to analysis implementations operate on. Our formalism is an imperative variant (with a call stack instead of continuations) of the corresponding formalism of Might, Smaragdakis and Van Horn [12], which attempts a unified treatment of control-flow analysis (in functional languages) and points-to analysis (in imperative/OO languages).

The grammar below describes A-Normal Featherweight Java. Some of the Featherweight Java conventions merit a reminder: A class declaration always names a superclass, and lists fields (distinct from those in the superclass) followed by a single constructor and a list of methods. Constructors are stylized, always taking in as many parameters as total fields in the class and superclasses, and consisting of a call to the superclass constructor and assignment of the rest of the fields, all in order.

$$\begin{aligned}
\mathsf{Class} &::= \texttt{class}\ C\ \texttt{extends}\ C'\ \{\overrightarrow{C''\ f\ ;}\ K\ \vec{M}\}\\
K \in \mathsf{Konst} &::= C\ (\overrightarrow{C\ f})\{\texttt{super}(\vec{f})\ ;\ \overrightarrow{\texttt{this}.f'' = f'''\,;}\}\\
M \in \mathsf{Method} &::= C\ m\ (\overrightarrow{C\ v})\ \{\ \overrightarrow{C\ v\ ;}\ \vec{s}\ \}\\
s \in \mathsf{Stmt} &::= v = e\ ;^{\ell} \mid \texttt{return}\ v\ ;^{\ell}\\
e \in \mathsf{Exp} &::= v \mid v.f \mid v.m(\vec{v}) \mid \texttt{new}\ C\ (\vec{v}) \mid (C)v
\end{aligned}$$

$v \in \mathsf{Var}$ is a set of variable names

$f \in \mathsf{FieldName}$ is a set of field names

$C \in \mathsf{ClassName}$ is a set of class names

$m \in \mathsf{MethodCall}$ is a set of method invocation sites

$\ell \in \mathsf{Lab}$ is a set of labels

Every statement has a label, to provide a convenient way of referencing program points. The function $succ : \mathsf{Lab} \rightharpoonup \mathsf{Stmt}$ yields the subsequent statement for a statement's label.

2.1 Concrete Semantics

We express the semantics using a small-step state machine. Figure 1 contains the state space. A state consists of a statement, a data stack (for local variables), a store, a call-stack (recording, for each active method invocation, the statement to return to, the context to restore, and the location that will hold the return value), and a current context. Following earlier work [12], our concrete semantics anticipates abstraction in that variables and fields are mapped to context-sensitive addresses, and the store maps such addresses to objects. The purpose served by the concept of a context-sensitive address is to introduce an extra level of indirection (through the store). In the usual concrete semantics, every dynamic instance of a variable or field will have a different context, making context-sensitive addresses extraneous. Nevertheless, the existence of the store makes it easy to collapse information from different program paths, as long as variables or fields map to the same context-sensitive address.[1] In addition to being a map from fields to context-sensitive addresses, an object also stores its creation context. There are two different kinds of context in this state space: a context for local (method) variables, and a heap context, for object fields. At a first approximation, one can think of the two contexts as being the same sets. Any infinite sets can play the role of context. By picking specific context sets we can simplify the mapping from concrete to abstract, as well as capture the essence of object-sensitivity.

The semantics are encoded as a small-step transition relation $(\Rightarrow) \subseteq \Sigma \times \Sigma$, shown in Figure 2. There is one transition rule for each expression type, plus an additional transition rule to account for return. Evaluation consists of finding all states reachable from an initial state (typically a single call statement with an empty store and binding environment). We use standard functions *cons*, *car*, *cdr*, and $first_n$ to construct and deconstruct lists/stacks. For convenience, we define a lookup of a variable in a stack to mean a lookup in the top component of the stack, e.g., $st(v)$ means $(car(st))(v)$. We also use helper functions

$$\mathcal{M} : Obj \times \mathsf{MethodCall} \rightharpoonup \mathsf{Method}$$
$$\mathcal{F} : \mathsf{ClassName} \to \mathsf{FieldName}^*.$$

[1] To prevent misunderstandings, we note that the extra level of indirection is the only purpose of the store in our semantics. Specifically, our store is not intended for modeling the Java heap and our stack is not modeling the Java local variable stack (although it has similar structure, as it is a map over local variables). For instance, the store is used to also map local variables to actual values, which is the purpose of a Java stack.

$$\begin{aligned}
\varsigma \in \Sigma &= \mathsf{Stmt} \times \mathit{Stack} \times \mathit{Store} \times \mathit{CallStack} \times \mathit{Context}\\
st \in \mathit{Stack} &= (\mathsf{Var} \rightharpoonup \mathit{ContSensAddr})^*\\
\sigma \in \mathit{Store} &= \mathit{ContSensAddr} \to \mathit{Obj}\\
o \in \mathit{Obj} &= \mathit{HContext} \times (\mathsf{FieldName} \rightharpoonup \mathit{ContSensAddr})\\
cst \in \mathit{CallStack} &= (\mathsf{Stmt} \times \mathit{Context} \times \mathit{ContSensAddr})^*\\
a \in \mathit{ContSensAddr} &= (\mathsf{Var} \times \mathit{Context}) + (\mathsf{FieldName} \times \mathit{HContext})
\end{aligned}$$

$c \in \mathit{Context}$ is an infinite set of contexts

$hc \in \mathit{HContext}$ is an infinite set of heap contexts.

Figure 1. State-space for A-Normal Featherweight Java.

Variable reference

$$([\![v = v'\ ;^{\ell}]\!], st, \sigma, cst, c) \Rightarrow (succ(\ell), st, \sigma', cst, c), \text{ where}$$
$$\sigma' = \sigma + [st(v) \mapsto \sigma(st(v'))].$$

Return

$$([\![\texttt{return}\ v\ ;^{\ell}]\!], st, \sigma, cst, c) \Rightarrow (s, cdr(st), \sigma', cdr(cst), c'), \text{ where}$$
$$(s, c', a_{ret}) = car(cst) \qquad \sigma' = \sigma + [a_{ret} \mapsto \sigma(st(v))].$$

Field reference

$$([\![v = v'.f\ ;^{\ell}]\!], st, \sigma, cst, c) \Rightarrow (succ(\ell), st, \sigma', cst, c), \text{ where}$$
$$(\_, [f \mapsto a_f]) = \sigma(st(v')) \qquad \sigma' = \sigma + [st(v) \mapsto \sigma(a_f)].$$

Method invocation

$$([\![v = v_0.m(\overrightarrow{v'})\ ;^{\ell}]\!], st, \sigma, cst, c) \Rightarrow (s_0, st', \sigma', cst', c'),$$
where
$$M = [\![C\ m\ (\overrightarrow{C\ v''})\ \{\overrightarrow{C'\ v'''\ ;}\ \vec{s}\}]\!] = \mathcal{M}(o_0, m)$$
$$o_0 = \sigma(st(v_0)) \qquad o_i = \sigma(st(v_i'))$$
$$(hc_0, \_) = o_0 \qquad c' = merge(\ell, hc_0, c)$$
$$a_i' = (v_i'', c') \qquad a_j'' = (v_j''', c')$$
$$\sigma' = \sigma + [a_i' \mapsto o_i] \qquad cst' = cons((succ(\ell), c, st(v)), cst)$$
$$st' = cons([v_i'' \mapsto a_i', v_j''' \mapsto a_j''], st).$$

Object allocation

$$([\![v = \texttt{new}\ C\ (\overrightarrow{v'})\ ;^{\ell}]\!], st, \sigma, cst, c) \Rightarrow (succ(\ell), st, \sigma', cst, c),$$
where
$$o_i = \sigma(st(v_i')) \qquad hc = record(\ell, c)$$
$$\vec{f} = \mathcal{F}(C) \qquad a_i = (f_i, hc)$$
$$o' = (hc, [f_i \mapsto a_i]) \qquad \sigma' = \sigma + [st(v) \mapsto o'] + [a_i \mapsto o_i].$$

Casting

$$([\![v = (C')\ v']\!], st, \sigma, cst, c) \Rightarrow (succ(\ell), st, \sigma', cst, c), \text{ where}$$
$$\sigma' = \sigma + [st(v) \mapsto \sigma(st(v'))].$$

Figure 2. Concrete semantics for A-Normal Featherweight Java.

The former takes a method invocation point and an object and returns the object's method that is called at that point (which is not known without knowing the receiver object, due to dynamic dispatch). The latter returns a list of all fields in a class definition, including superclass fields.

Our semantics is parameterized by two functions that manipulate contexts:

$$\mathit{record} : \mathsf{Lab} \times \mathit{Context} \to \mathit{HContext}$$
$$\mathit{merge} : \mathsf{Lab} \times \mathit{HContext} \times \mathit{Context} \to \mathit{Context}$$

The *record* function is used every time an object is created, in order to store a creation context with the object. The *merge* function is used on every method invocation. Its first argument is the current call statement label, while the second and third arguments are the context of allocation of the method's receiver object and the current (caller's) context, respectively. The key for different flavors of context sensitivity is to specify different *record* and *merge* functions for contexts.

For a simple understanding of the concrete semantics that yields the natural behavior one expects, we can define contexts as just natural numbers:

$$\mathit{Context} = \mathit{HContext} = \mathbb{N}.$$

In that case, we need to ensure that the *record* and *merge* functions never return a duplicate context. For instance, if we consider labels to also map to naturals we can capture the entire history of past context creation by defining $\mathit{record}(\ell, c) = 2^\ell \cdot 3^c$ and $\mathit{merge}(\ell, hc, c) = 5^\ell \cdot 7^{hc} \cdot 11^c$. A different choice of context that is closer to idealized (infinite context) object-sensitive semantics consists of defining a context as a list of labels:

$$\mathit{Context} = \mathit{HContext} = \mathsf{Lab}^*,$$

yielding a straightforward *record* function, while the *merge* function can ignore its second argument:

$$\mathit{record}(\ell, c) = \mathit{cons}(\ell, c)$$
$$\mathit{merge}(\ell, hc, c) = \mathit{cons}(\ell, c).$$

These definitions enable every variable to have a different address for different invocations, and every object field to have a different address for each object allocated, as expected.[2]

2.2 Abstract Semantics

It is now straightforward to express object-sensitive points-to analyses by abstract interpretation [4] of the above semantics for an abstract domain that maintains only finite context.

The abstract state space is shown in Figure 3. The main distinctions from the concrete state-space are that the set of contexts is finite and that the store can return a *set* of objects, instead of a single object. Generally, the abstract semantics closely mirror the concrete.

The abstract semantics are encoded as a small-step transition relation $(\rightsquigarrow) \subseteq \hat{\Sigma} \times \hat{\Sigma}$, shown in Figure 4. There is one abstract transition rule for each expression type, plus an additional transition rule to account for return. We assume the usual properties for $\sqcup$ of a map to sets (i.e., merging of the sets for the same value).

We similarly define abstract versions of the context-manipulating functions:

$$\widehat{\mathit{record}} : \mathsf{Lab} \times \widehat{\mathit{Context}} \to \widehat{\mathit{HContext}}$$
$$\widehat{\mathit{merge}} : \mathsf{Lab} \times \widehat{\mathit{HContext}} \times \widehat{\mathit{Context}} \to \widehat{\mathit{Context}}$$

[2] Technically, this is true only because the FJ calculus has no iteration, so object allocations from the same statement can only occur after a recursive call. Thus, the string of labels for method calls is enough to ensure that objects have a unique context.

$$\hat{\varsigma} \in \hat{\Sigma} = \mathsf{Stmt} \times \widehat{\mathit{Stack}} \times \widehat{\mathit{Store}} \times \widehat{\mathit{CallStack}} \times \widehat{\mathit{Context}}$$
$$\widehat{st} \in \widehat{\mathit{Stack}} = (\mathsf{Var} \rightharpoonup \widehat{\mathit{ContSensAddr}})^*$$
$$\hat{\sigma} \in \widehat{\mathit{Store}} = \widehat{\mathit{ContSensAddr}} \to \mathcal{P}\left(\widehat{\mathit{Obj}}\right)$$
$$\hat{o} \in \widehat{\mathit{Obj}} = \widehat{\mathit{HContext}} \times (\mathsf{FieldName} \rightharpoonup \widehat{\mathit{ContSensAddr}})$$
$$\widehat{cst} \in \widehat{\mathit{CallStack}} = (\mathsf{Stmt} \times \widehat{\mathit{Context}} \times \widehat{\mathit{ContSensAddr}})^*$$
$$\hat{a} \in \widehat{\mathit{ContSensAddr}} = (\mathsf{Var} \times \widehat{\mathit{Context}}) + (\mathsf{FieldName} \times \widehat{\mathit{HContext}})$$
$$\hat{c} \in \widehat{\mathit{Context}} \text{ is a finite set of contexts}$$
$$\widehat{hc} \in \widehat{\mathit{HContext}} \text{ is a finite set of heap contexts.}$$

Figure 3. Object-sensitive analysis state-space for A-Normal Featherweight Java.

Variable reference

$$([\![v = v'\,;^\ell]\!], \widehat{st}, \hat{\sigma}, \widehat{cst}, \hat{c}) \rightsquigarrow (\mathit{succ}(\ell), \widehat{st}, \hat{\sigma}', \widehat{cst}, \hat{c}), \text{ where}$$
$$\hat{\sigma}' = \hat{\sigma} \sqcup [\widehat{st}(v) \mapsto \hat{\sigma}(\widehat{st}(v'))].$$

Return

$$([\![\texttt{return}\ v\ ;^\ell]\!], \widehat{st}, \hat{\sigma}, \widehat{cst}, \hat{c}) \rightsquigarrow (s, \mathit{cdr}(\widehat{st}), \hat{\sigma}', \mathit{cdr}(\widehat{cst}), \hat{c}'), \text{ where}$$
$$(s, \hat{c}', \hat{a}_{ret}) = \mathit{car}(\widehat{cst}) \qquad \hat{\sigma}' = \hat{\sigma} \sqcup [\hat{a}_{ret} \mapsto \hat{\sigma}(\widehat{st}(v))].$$

Field reference

$$([\![v = v'.f\ ;^\ell]\!], \widehat{st}, \hat{\sigma}, \widehat{cst}, \hat{c}) \rightsquigarrow (\mathit{succ}(\ell), \widehat{st}, \hat{\sigma}', \widehat{cst}, \hat{c}), \text{ where}$$
$$(\_, [f \mapsto \hat{a}_f]) = \hat{\sigma}(\widehat{st}(v')) \qquad \hat{\sigma}' = \hat{\sigma} \sqcup [\widehat{st}(v) \mapsto \hat{\sigma}(\hat{a}_f)].$$

Method invocation

$$([\![v = v_0.m(\overrightarrow{v'})\,;^\ell]\!], \widehat{st}, \hat{\sigma}, \widehat{cst}, \hat{c}) \rightsquigarrow (s_0, \widehat{st}', \hat{\sigma}', \widehat{cst}', \hat{c}'),$$
where
$$M = [\![C\ m\ (\overrightarrow{C\ v''})\ \{\overrightarrow{C'\ v'''\ ;}\ \vec{s}\}]\!] = \mathcal{M}(\hat{o}_0, m)$$
$$\hat{o}_0 \in \hat{\sigma}(\widehat{st}(v_0)) \qquad \hat{o}_i = \hat{\sigma}(\widehat{st}(v_i'))$$
$$(\widehat{hc}_0, \_) = \hat{o}_0 \qquad \hat{c}' = \widehat{\mathit{merge}}(\ell, \widehat{hc}_0, \hat{c})$$
$$\hat{a}_i' = (v_i'', \hat{c}') \qquad \hat{a}_j'' = (v_j''', \hat{c}')$$
$$\hat{\sigma}' = \hat{\sigma} \sqcup [\hat{a}_i' \mapsto \hat{o}_i] \qquad \widehat{cst}' = \mathit{cons}((\mathit{succ}(\ell), \hat{c}, \widehat{st}(v)), \widehat{cst})$$
$$\widehat{st}' = \mathit{cons}([v_i'' \mapsto \hat{a}_i', v_j''' \mapsto \hat{a}_j''], \widehat{st}).$$

Object allocation

$$([\![v = \texttt{new}\ C\ (\overrightarrow{v'})\,;^\ell]\!], \widehat{st}, \hat{\sigma}, \widehat{cst}, \hat{c}) \rightsquigarrow (\mathit{succ}(\ell), \widehat{st}, \hat{\sigma}', \widehat{cst}, \hat{c}),$$
where
$$\hat{o}_i = \hat{\sigma}(\widehat{st}(v_i')) \qquad \widehat{hc} = \widehat{\mathit{record}}(\ell, \hat{c})$$
$$\vec{f} = \mathcal{F}(C) \qquad \hat{a}_i = (f_i, \widehat{hc})$$
$$\hat{o}' = (\widehat{hc}, [f_i \mapsto \hat{a}_i]) \qquad \hat{\sigma}' = \hat{\sigma} \sqcup [\widehat{st}(v) \mapsto \hat{o}'] \sqcup [\hat{a}_i \mapsto \hat{o}_i].$$

Casting

$$([\![v = (C')\ v']\!], \widehat{st}, \hat{\sigma}, \widehat{cst}, \hat{c}) \rightsquigarrow (\mathit{succ}(\ell), \widehat{st}, \hat{\sigma}', \widehat{cst}, \hat{c}), \text{ where}$$
$$\hat{\sigma}' = \hat{\sigma} \sqcup [\widehat{st}(v) \mapsto \hat{\sigma}(\widehat{st}(v'))].$$

Figure 4. Object-sensitivity abstract semantics for A-Normal Featherweight Java.

The abstract $\widehat{record}$ and $\widehat{merge}$ functions capture the essence of object-sensitivity: an object-sensitive analysis is distinguished by its storing a context together with every allocated object (via $\widehat{record}$), and by its retrieving that context and using it as the basis for the analysis context of every method dispatched on the object (via $\widehat{merge}$).

2.3 Analysis Variations and Observations

With the above framework, we can concisely characterize all past object-sensitive analyses, as well as discuss other possibilities. All variations consist of only modifying the definitions of $\widehat{Context}$, $\widehat{HContext}$, $\widehat{record}$, and $\widehat{merge}$.

Original Object Sensitivity. Milanova et al. [13] gave the original definition of object-sensitivity but did not implement the analysis for context depths greater than 1. Taken literally, the original definition prescribes that, for an n-object-sensitive analysis, the regular and the heap context consist of n labels:

$$\widehat{Context} = \widehat{HContext} = \mathsf{Lab}^n,$$

while $\widehat{record}$ just keeps the first n elements of the context defined in our concrete semantics, and $\widehat{merge}$ discards everything but the receiver object context:

$$\widehat{record}(\ell, \hat{c}) = cons(\ell, first_{n-1}(\hat{c}))$$
$$\widehat{merge}(\ell, \widehat{hc}, \hat{c}) = \widehat{hc}.$$

In practical terms, this definition has several consequences:

- The only labels used as context are labels from an object allocation site, via the $\widehat{record}$ function.
- On a method invocation, only the (heap) context of the receiver object matters.
- The heap context of a newly allocated object is derived from the context of the method doing the allocation, i.e., from the heap context of the object that is doing the allocation.

In other words, the context used to analyze a method consists of the allocation site of the method's receiver object, the allocation site of the object that allocated the method's receiver object, the allocation site of the object that allocated the object that allocated the method's receiver object, and so on. In the next section we discuss whether this is a good choice of context for scalability and precision.

Past Implementations of Object Sensitivity. For an object-sensitive analysis with context depth 1, the context choice is obvious. The $\widehat{merge}$ function has to use some of the receiver object context, or the analysis would not be object-sensitive (just call-site-sensitive), and the receiver object context consists of just the object's allocation site.

For analyses with context depth $n > 1$, however, the definitions of $\widehat{merge}$ and $\widehat{record}$ can vary significantly. Actual implementations of such analyses deviated from the Milanova et al. definition. Notably, the Paddle framework [7, 9] (which provides the most-used such implementation available) merges the allocation site of the receiver object with multiple context elements *from the caller object context* when analyzing a method. This results in the following functions:

$$\widehat{record}(\ell, \hat{c}) = cons(\ell, first_{n-1}(\hat{c}))$$
$$\widehat{merge}(\ell, \widehat{hc}, \hat{c}) = cons(car(\widehat{hc}), first_{n-1}(\hat{c})).$$

The practically interesting case is of $n = 2$. (Higher values are well outside current technology if the analysis is applied as defined, i.e., the context depth applies to all program variables.) The above definition then means that every method is analyzed using as context a) the allocation site of the receiver object; and b) the allocation site of the caller object.

Heap Context, Naming, and Full-Object Sensitivity. The above analyses defined the heap context ($\widehat{HContext}$) and the regular/method context ($\widehat{Context}$) to be the same set, namely Lab^n. There are other interesting possibilities, however. Since the $\widehat{merge}$ function has access to a heap context and to a regular context (and needs to build a new regular context) the heap context can be shallower. For instance, Lhoták's exploration of object-sensitive analyses [7] studies in depth analyses where the heap context is always just a single allocation site:

$$\widehat{HContext} = \mathsf{Lab}$$
$$\widehat{Context} = \mathsf{Lab}^n$$
$$\widehat{record}(\ell, \hat{c}) = \ell$$
$$\widehat{merge}(\ell, \widehat{hc}, \hat{c}) = cons(\widehat{hc}, first_{n-1}(\hat{c})).$$

In fact, *the above definition is what is most commonly called an "object-sensitive analysis" in the literature*! The analyses we saw earlier are colloquially called "object-sensitive analyses with a context-sensitive (or object-sensitive) heap". That is, the points-to analysis literature by convention uses context to apply only to methods, while heap objects are represented by just their allocation site. Adding more context to object fields than just the object allocation site is designated with suffixes such as "context-sensitive heap" or "heap cloning" in an analysis description. Thus, one needs to be very careful with naming conventions. We summarize ours at the end of this section.

Another interesting possibility is that of keeping a *deeper* context for heap objects than for methods. The most meaningful case in practice is the one where the heap object keeps one extra context element:

$$\widehat{HContext} = \mathsf{Lab}^{n+1}$$
$$\widehat{Context} = \mathsf{Lab}^n$$
$$\widehat{record}(\ell, \hat{c}) = cons(\ell, \hat{c}).$$

(The $\widehat{merge}$ function can vary orthogonally, as per our earlier discussion.) For $n = 1$, this is Lhoták's "1obj+H" analysis, which is currently considered the best trade-off between scalability and precision [8], and to which we refer repeatedly in our experimental evaluation of Section 5

This latest variation complicates naming even more. In Lhoták's detailed naming scheme, the above analysis would be an "n-object-sensitive analysis with an (n-)object-sensitive heap", while our standard n-object-sensitive analysis (with $\widehat{Context} = \widehat{HContext} = \mathsf{Lab}^n$) is called an "$n$-object-sensitive analysis with an $n-1$-object-sensitive heap". The reason for the off-by-one convention is historical: it is considered self-evident in the points-to analysis community that the static abstraction for a heap object will consist of at least its allocation site label. Therefore, the heap context is considered to be $n-1$ labels, when the static abstraction of an object consists of n labels in total.

In the rest of this paper, we adopt the standard terminology of the points-to analysis literature. That is, we talk of an "n-object-sensitive analysis with an $n-1$-object-sensitive heap" to mean that $\widehat{Context} = \widehat{HContext} = \mathsf{Lab}^n$. Furthermore, to distinguish between the two dominant definitions of a $\widehat{merge}$ function (Milanova et al.'s, as opposed to that of the Paddle framework) we refer to a *full-object-sensitive analysis* vs. a *plain-object-sensitive analysis*. A full-object-sensitive analysis is characterized by a $\widehat{merge}$ function:

$$\widehat{merge}(\ell, \widehat{hc}, \hat{c}) = \widehat{hc}.$$

That is, the "full" object abstraction of the receiver object is used as context for the method invoked on that object. In contrast, a plain-object-sensitive analysis merges information from the receiver and the caller objects:

$$\widehat{record}(\ell, \hat{c}) = cons(\ell, first_{n-1}(\hat{c}))$$

$$\widehat{merge}(\ell, \widehat{hc}, \hat{c}) = cons(car(\widehat{hc}), first_{n-1}(\hat{c})).$$

Thus, the original object-sensitivity definition by Milanova et al. is a full-object-sensitive analysis, while the practical object-sensitive analysis of the Paddle framework is plain-object-sensitive. We abbreviate plain-object-sensitive analyses for different context and heap context depths to "*n*plain+*m*H" and full-object-sensitive analyses to "*n*full+*m*H". When the two analyses coincide, we use the abbreviation "*n*obj+*m*H".

Discussion. Finally, note that our theoretical framework is not limited to traditional object-sensitivity, but also encompasses call-site sensitivity. Namely, the $\widehat{merge}$ function takes the current call-site label as an argument. None of the actual object-sensitive analyses we saw above use this argument. Thus, our framework suggests a generalized interpretation of what constitutes an object-sensitive analysis. We believe that the essence of object-sensitivity is *not* in "only using allocation-site labels as context" but in "storing an object's creation context and using it at the site of a method invocation" (i.e., the functionality of the $\widehat{merge}$ function). We expect that future analyses will explore this direction further, possibly combining call-site and allocation-site information in interesting ways.

Our framework also allows the same high-level structure of an analysis but with different context information preserved. Section 4 pursues a specific direction in this design space, but generally we can produce analyses by using as context *any abstractions computable from the arguments to functions* $\widehat{record}$ *and* $\widehat{merge}$ (with appropriate changes to the $\widehat{Context}$ and $\widehat{HContext}$ sets). For instance, we can use as context coarse-grained program location information (module identifiers, packages, user annotations) since these are uniquely identified by the current program statement label, ℓ. Similarly, we can make different context choices for different allocation sites or call sites, by defining the $\widehat{record}$ and $\widehat{merge}$ functions conditionally on the supplied label. In fact, there are few examples of context-sensitive points-to analyses that cannot be captured by our framework, and simple extensions would suffice for most of those. For instance, a context-sensitive points-to analysis that employs as context a static abstraction of the arguments of a method call (and not just of the receiver object) is currently not directly expressible in our formalism. (This approach has been fruitful in different static analyses, e.g., for type inference [1, 15].) Nevertheless, it is a simple matter to adjust the form of the $\widehat{merge}$ function and the "method invocation" rule in order to allow the method call context to also be a function of the heap contexts of argument objects.

3. Insights on Context Choice

With the benefit of our theoretical map of context choices for object-sensitivity, we next discuss what makes a scalable and precise analysis in practice.

3.1 Full-Object-Sensitivity vs. Plain-Object-Sensitivity

The first question in our work is how to select which context elements to keep—i.e., what is the best definition of the $\widehat{merge}$ function for practical purposes. We already saw the two main distinctions, in the form of full-object-sensitive vs. plain-object-sensitive analyses. Consider the standard case of a 2-object-sensitive analysis with a 1-object-sensitive heap, i.e., per the standard naming convention, $\widehat{Context} = \widehat{HContext} = \mathsf{Lab}^2$. The 2full+1H analysis will examine every method using as context the allocation site of the method's receiver object, and the allocation site of the allocator of this receiver object. (Recall that all information for method invocations under the same context will be merged, while information under different contexts will be kept separate.) In contrast the 2plain+1H analysis examines every method using as context the allocation site of the receiver object, and the allocation site of the caller object.

The 2full+1H analysis has not been implemented or evaluated in practice before. Yet with an understanding of how context is employed, there are strong conceptual reasons why one may expect 2full+1H to be superior to 2plain+1H. The insight is that context serves the purpose of yielding extra information to classify dynamic paths, at high extra cost for added context depth. Thus, for context to serve its purpose, it needs to be a good classifier, splitting the space of possibilities in roughly uniform partitions. When mixing context elements (allocation site labels) from the receiver and the caller object (as in the 2plain+1H analysis), the two context elements are likely to be correlated. High correlation means that a 2-object-sensitive analysis is effectively reduced to a high-cost 1-object-sensitive one. In a simple case of context correlation, an object calls another method on itself: the receiver object and the caller object are the same.[3] There are many more patterns of common object correlation—knowing that we are executing a method of object p almost always yields significant information about which object q is the receiver of a method call. Wrapper patterns, proxy patterns, the Bridge design pattern, etc., all have pairs of objects that are allocated and used together. For such cases of related objects, one can see the effect in intuitive terms: The traditional mixed context of a 2plain+1H analysis classifies method calls by asking objects "where were you born and where was your sibling born?" A 2full+1H analysis asks "where were you born and where was your parent born?" The latter is a better differentiator, since birth locations of siblings are more correlated.

3.2 Context Depth and Analysis Complexity

The theme of context element correlation and its effect on precision is generally important for analysis design. The rule of thumb is that context elements should be as little-correlated as possible for an analysis with high precision. To see this consider how context depth affects the scalability of an analysis. There are two opposing forces when context depth is increased. On the one hand, increased precision may help the analysis avoid combinatorial explosion. On the other hand, when imprecision will occur anyway, a deeper context analysis will be significantly less scalable.

For the first point, consider the question "when can an analysis with a deeper context outperform one with a shallower context?" Concretely, are there cases when a 2-object-sensitive analysis will be faster or more scalable than a 1-object-sensitive analysis (for otherwise the same analysis logic, i.e., both plain-obj or full-obj)? Much of the cost of evaluating an analysis is due to propagating matching facts. Consider, for instance, the "Variable reference" rule from our abstract semantics in Figure 4, which results in an evaluation of the form: $\hat{\sigma}' = \hat{\sigma} \sqcup [\widehat{st}(v) \mapsto \hat{\sigma}(\widehat{st}(v'))]$. For this evaluation to be faster under a deeper context, the lookup $\hat{\sigma}(\widehat{st}(v'))$ should return substantially fewer facts, i.e., the analysis context should result in much higher precision. Specifically two conditions need to be satisfied. First, the more detailed context should partition the facts well: the redundancy should be minimal between partitions in context-depth-2 that would project to the same partition with context depth 1. Intuitively, adding an extra level of context should not cause all (or many) facts to be replicated

[3] This case can be detected statically by the analysis, and context repetition can be avoided. This simple fix alone is not sufficient for reliably reducing the imprecision of a 2plain+1H analysis, however.

for all (or many) extra context elements. Second, fewer facts should be produced by the rule evaluation at context depth 2 relative to depth 1, when compared after projection down to depth-1 facts. In other words, going to context depth 2 should be often enough to tell us that some object does not in fact flow to a certain 1-context-sensitive variable, because the assignment is invalid given the more precise context knowledge.

In case of imprecision, on the other hand, the deeper context will almost always result in a combinatorial explosion of the possible facts and highly inefficient computation. When going from a 1obj+H analysis to a 2full+H or 2plain+H, we have every fact analyzed in up to N times more contexts (where N is the total number of context elements, i.e., allocation sites). As a rule of thumb, every extra level of context can multiply the space consumption and runtime of an analysis by a factor of N, and possibly more, since the N-times-larger collections of facts need to be used to index in other N-times-larger collections, with indexing mechanisms (and possibly results) that may not be linear.

It is, therefore, expected that an analysis with deeper context will perform quite well when it manages to keep precise facts, while exploding in runtime complexity when the context is not sufficient to maintain precision. Unfortunately, there are inevitable sources of imprecision in any real programming language—these include reflection (which is impossible to handle soundly and precisely), static fields (which defeat context-sensitivity), arrays, exceptions, etc. When this imprecision is not well-isolated and affects large parts of a realistic program, the points-to analysis will almost certainly fail (within any reasonable time and space bound). This produces a *scalability wall* effect: a given analysis on a program will either terminate quickly or will fail to terminate "ever" (in practical terms). Input characteristics of the program (e.g., size metrics) are almost never good predictors of whether the program will be easy to analyze, as this property depends directly on the induced analysis imprecision.

The challenge then becomes whether we can maintain high precision while reducing the possibility for a combinatorial blowup of analysis facts due to deeper context. We next introduce a new approach in this direction.

4. Type-Sensitivity

If an analysis with a deeper context by nature results in a combinatorial explosion in complexity, then a natural step is to reduce the base of the exponential function. Context elements in an object-sensitive analysis are object allocation sites, and typical programs have too many allocation sites, making the product of the number of allocation sites too high. Therefore, a simple idea for scalability is to use coarser approximations of objects as context, instead of complete allocation sites. This would collapse the possible combinations down to a more manageable space, yielding improved scalability. The most straightforward static abstraction that can approximate an object is a type, which leads to our idea of a *type-sensitive* analysis.

4.1 Definition of Type-Sensitive Analysis

A type-sensitive analysis is almost identical to an object-sensitive analysis, but *whereas an object-sensitive analysis would keep an allocation site as a context element, a type-sensitive analysis keeps a type instead*.[4] Consider, for instance, a 2-type-sensitive analysis with a 1-type-sensitive heap (henceforth "2type+1H"). The method context for this analysis consists of two types. (For now we do not care which types. We discuss later what types yield high precision.)

[4] Despite the name similarity, our type-sensitive points-to analysis has no relationship to Reppy's "type-sensitive control-flow analysis" [16], which uses types to filter control flow facts in a context-insensitive analysis.

Expressed in our framework, the 2type+1H analysis has:

$$\widehat{HContext} = \mathsf{Lab} \times \mathsf{ClassName}$$
$$\widehat{Context} = \mathsf{ClassName}^2.$$

Note again the contrast of the standard convention of the points-to analysis community and the structure of abstractions: the 2type+1H analysis also includes an allocation site label in the static abstraction of an object—this aspect is considered so essential that it is not reflected on the analysis name. Approximating the allocation site of the object itself by a type would be possible, but detrimental for precision.

Generally, type-contexts and object-contexts can be merged at any level, as long as the $\widehat{merge}$ function can be defined. An interesting choice is an analysis that merely replaces one of the two allocation sites of 2full+1H with a type, while leaving all the rest of the context elements intact. We call this a 1-type-1-object-sensitive analysis with a 1-object-sensitive heap, and shorten the name to "1type1obj+1H". That is, a 1type1obj+1H analysis has:

$$\widehat{HContext} = \mathsf{Lab}^2$$
$$\widehat{Context} = \mathsf{Lab} \times \mathsf{ClassName}.$$

The 2type+1H and the 1type1obj+1H analyses are the most practically promising type-sensitive analyses with a context depth of 2. Their context-manipulation functions can be described with the help of an auxiliary function

$$\mathcal{T} : \mathsf{Lab} \to \mathsf{ClassName},$$

which retrieves a type from an allocation site label. For 2type+1H, the context functions become:[5]

$$\widehat{record}(\ell, \hat{c} = [C_1, C_2]) = [\ell, C_1]$$
$$\widehat{merge}(\ell, \widehat{hc} = [\ell', C], \hat{c}) = [\mathcal{T}(\ell'), C],$$

while for 1type1obj+1H, the two functions are:

$$\widehat{record}(\ell, \hat{c} = [\ell', C]) = [\ell, \ell']$$
$$\widehat{merge}(\ell, \widehat{hc} = [\ell_1, \ell_2], \hat{c}) = [\ell_1, \mathcal{T}(\ell_2)],$$

In other words, the two analyses are variations of 2full+1H (and not of 2plain+1H), with some of the context information downgraded to be types instead of allocation site labels. The function $\mathcal{T}$ makes opaque the method we use to produce a type from an allocation site, which we discuss next.

4.2 Choice of Type Contexts

Just having a type as a context element does not tell us how good the context will be for ensuring precision—the choice of type is of paramount importance. The essence of understanding what constitutes a good type context is the question "what does an allocation site tell us about types?" After all, we want to use types as a coarse approximation of allocation sites, so we want to maintain most of the information that allocation sites imply regarding types.

The identity of an allocation site, i.e., an instruction "`new A()`" inside class `C`, gives us:

- the dynamic type `A` of the allocated object
- an upper bound `C` on the dynamic type of the *allocator* object. (Since the allocation site occurs in a method of class `C`, the allocator object must be of type `C` or a subclass of `C` that does not override the method containing the allocation site.)

[5] We use common list and pattern-matching notation to avoid long expressions. E.g., "$\widehat{record}(\ell, \hat{c} = [C_1, C_2])$" means "when the second argument, $\hat{c}$, of $\widehat{record}$ is a list of two elements, C_1 and C_2...".

A straightforward option would be to define the $\mathcal{T}$ function to return just the type of the allocation site, i.e., type `A` above. This is an awful design decision, however. To see why, consider first the case of a 2type+1H analysis. When we analyze a method context-sensitively and the first element of the context is the type of the receiver object, we are effectively wasting most of the potential of the context. The reason is that the method under analysis already gives us enough information about the type of the receiver object—i.e., the identity of the method and the type of the receiver are closely correlated. If, for instance, the method being analyzed is "`B::foo`" (i.e., method `foo` defined in class `B`) then we already have a tight upper bound on the dynamic type of the receiver object: the receiver object's type has to be either `B` or a subclass of `B` that does not override method `foo`. Since we want to pick a context that is less correlated with other information and yields meaningful distinctions, a 2type+1H analysis should have its $\mathcal{T}$ function return the type in which the allocation takes place, i.e., class `C` above.

A similar argument applies to a 1type1obj+1H analysis. In this analysis the method context consists of the receiver object, as well as a type. We want 1type1obj+1H to be a good approximation of 2full+1H, which would keep two allocation sites instead (that of the receiver object and that of the receiver object's allocator object). Thus the two allocation sites of 2full+1H give us the following information about types:

- the dynamic type of the receiver object
- an upper bound on the dynamic type of the receiver object's allocator object
- the dynamic type of the receiver object's allocator object
- an upper bound on the dynamic type of the receiver object's allocator's allocator object.

The first two pieces of information, above, come from the identity of the receiver object's allocation site, and, thus, are kept intact in a 1type1obj+1H analysis. The question is which of the last two types we would like to keep. The high correlation of the second and third bullet point above (upper bound of a type and the type itself) makes it clear that we want to keep the type of the last bullet. That is, in all scenarios, the function $\mathcal{T}(l)$, when l represents an instruction "`new A()`" inside class `C`, should return type `C` and not type `A`. We validate this understanding experimentally as part of the results of the next section.

5. Implementation and Evaluation

We implemented and evaluated several object-sensitive analyses for a context depth up to 2, which meets the limit of practicality for real-world programs. The questions we want to answer relate to the main new ideas presented so far:

- Is full-object-sensitivity advantageous compared to plain-object-sensitivity in terms of precision and performance, as argued in Section 3.1? (Recall that full-object-sensitive analyses had not been implemented in the past for context depth greater than 1.)
- Does the definition of function $\mathcal{T}$ matter, as predicted in Section 4.2?
- Does type-sensitivity achieve higher scalability than regular object-sensitive analyses while maintaining most of the precision?

5.1 Setting

Our implementation is in the context of the Doop framework [2, 3]. Doop uses the Datalog language to specify analyses declaratively. Additionally, Doop employs an *explicit* representation of relations, listing all the elements of tuples of related elements explicitly, as opposed to employing Binary Decision Diagrams (BDDs), which have often been used in points-to analysis [7, 8, 22, 23]. As we showed in earlier work [3] BDDs are only useful when the selected context abstractions introduce high redundancy, while analyses that take care to avoid unnecessary imprecision are significantly faster and scalable in an explicit representation. Doop is a highly scalable framework and implements very efficiently the most complex and precise context-sensitive analyses in current use [3]. Doop achieves functional equivalence (identical results) with Lhoták and Hendren's Paddle system [8], which is another feature-rich framework for precise analyses, but based on an entirely different architecture (using semi-declarative analysis specifications and BDDs to represent relations). This equivalence of results is useful for establishing that an analysis is correct and meaningful, which is a property far from granted for complex points-to analysis algorithms.

We use a 64-bit machine with a quad-core Xeon E5530 2.4GHz CPU (only one thread was active at a time). The machine has 24GB of RAM, but we have found no analysis that terminates (within two hours, but also occasionally allowing up to 12) after occupying more than 12GB of memory. Most analyses require less than 2GB, with only the longest running analyses (over 1000 seconds run time) occasionally needing more memory. This is indicative of the scalability wall described earlier: the explosion of contexts without a corresponding increase in precision makes an analysis intractable quite abruptly.

We analyzed the DaCapo benchmark programs, v.2006-10-MR2, with JDK 1.4 (j2re1.4.2_18). These benchmarks are the largest in the literature on context-sensitive points-to analysis.

We analyzed all benchmarks except hsqldb and jython with the full Doop functionality and support for language features, including native methods, reflection (a refinement of Livshits et al.'s algorithm [11] with support for reflectively invoked methods and constructors), and precise exception handling [2]. Generally our settings are a superset (i.e., more complete feature support) than prior published benchmarks on Doop [2, 3]. (The Doop language feature support is among the most complete in the literature, as documented in detail in the past [3].) Hsqldb and jython could not be analyzed with reflection analysis enabled—hsqldb cannot even be analyzed context-insensitively and jython cannot even be analyzed with the 1obj analysis. This is due to vast imprecision introduced when reflection methods are not filtered in any way by constant strings (for classes, fields, or methods) and the analysis infers a large number of reflection objects to flow to several variables. (E.g., in the theoretically worst case 244 sites in jython can be inferred to allocate over 1.1 million abstract objects.) For these two applications, our analysis has reflection reasoning disabled. Since hsqldb in the DaCapo benchmark code has its main functionality called via reflection, we had to configure its entry point manually.

5.2 Full-object-sensitivity vs. Plain-object-sensitivity

Figure 5 shows the precision comparison of a 2plain+1H and a 2full+1H analysis for a subset of the DaCapo benchmarks. For reference, we also include a context-insensitive, 1-object-sensitive and 1obj+1H analysis, and indicate how the metrics change from an analysis to the next more precise one. The metrics are a mixture of core points-to statistics and client analysis metrics, resembling the methodology of Lhoták and Hendren [8]. For ease of reference, we highlight (in bold) some of the most important metrics: the number of methods inferred to be reachable (including both application methods and methods in the standard Java library), the average var-points-to set (i.e., how many allocation sites a variable can refer to), the total number of call sites that are found to be polymorphic (i.e., for which the analysis cannot pinpoint a single method as the target of the dynamic dispatch), and the total number of casts that

		insensitive	1obj	1obj+H	2plain+1H	2full+1H
antlr	call-graph edges	43055	-559	-1216	-1129	-368
	reachable methods	**5758**	**-29**	**-37**	**-62**	**-21**
	total reachable virtual call sites	27823	-128	-96	-272	-139
	total polymorphic call sites	**1326**	**-38**	**-22**	**-38**	**-68**
	application reachable virtual call sites	16393	0	0	0	-9
	application polymorphic call sites	851	0	0	0	0
	total reachable casts	1038	-14	-15	-33	-6
	total casts that may fail	**844**	**-136**	**-94**	**-144**	**-64**
	application reachable casts	308	0	0	0	-1
	application casts that may fail	262	-8	-38	-66	-23
	average var-points-to	**216.71**	**24.7**	**15.1**	**8.5**	**8.2**
	average application var-points-to	327.27	20.8	15.3	8.8	8.5
chart	call-graph edges	44930	-1239	-2063	-2287	-765
	reachable methods	**8502**	**-76**	**-87**	**-115**	**-53**
	total reachable virtual call sites	23944	-233	-327	-368	-172
	total polymorphic call sites	**1218**	**-90**	**-24**	**-83**	**-119**
	application reachable virtual call sites	3649	0	-8	-47	-12
	application polymorphic call sites	110	-4	-13	-10	-4
	total reachable casts	1728	-22	-38	-58	-7
	total casts that may fail	**1457**	**-182**	**-252**	**-164**	**-120**
	application reachable casts	232	0	-4	-21	-1
	application casts that may fail	196	-17	-64	-32	-38
	average var-points-to	**98.35**	**36.0**	**20.1**	**9.4**	**6.7**
	average application var-points-to	55.35	27.2	14.4	5.0	2.8
eclipse	call-graph edges	36057	-1342	-2499	-1900	-803
	reachable methods	**6541**	**-67**	**-153**	**-83**	**-49**
	total reachable virtual call sites	18447	-291	-315	-356	-226
	total polymorphic call sites	**873**	**-96**	**-25**	**-38**	**-89**
	application reachable virtual call sites	5959	-101	-53	-31	-44
	application polymorphic call sites	292	-40	-9	-3	-3
	total reachable casts	1270	-21	-21	-27	-24
	total casts that may fail	**1001**	**-213**	**-103**	**-133**	**-51**
	application reachable casts	476	-3	-2	-8	0
	application casts that may fail	362	-61	-55	-58	1
	average var-points-to	**102.7**	**21.8**	**16.1**	**9.8**	**9.1**
	average application var-points-to	104.1	22.9	15.8	9.6	9.4
luindex	call-graph edges	24069	-663	-1155	-1119	-354
	reachable methods	**4742**	**-31**	**-34**	**-64**	**-20**
	total reachable virtual call sites	12675	-193	-117	-276	-130
	total polymorphic call sites	**507**	**-50**	**-22**	**-38**	**-66**
	application reachable virtual call sites	1267	-65	-29	-4	0
	application polymorphic call sites	39	-10	0	0	0
	total reachable casts	790	-14	-15	-33	-5
	total casts that may fail	**627**	**-129**	**-69**	**-96**	**-43**
	application reachable casts	63	0	-1	0	0
	application casts that may fail	46	-3	-18	-13	-2
	average var-points-to	**69.66**	**14.7**	**10.8**	**6.5**	**6.3**
	average application var-points-to	90.05	9.2	5.9	3.0	2.9
pmd	call-graph edges	30990	-527	-1329	-809	-1043
	reachable methods	**6158**	**-34**	**-44**	**-37**	**-62**
	total reachable virtual call sites	16029	-144	-163	-208	-280
	total polymorphic call sites	**576**	**-42**	**-26**	**-21**	**-107**
	application reachable virtual call sites	4545	-22	-87	-5	-83
	application polymorphic call sites	93	-2	-10	-4	-24
	total reachable casts	1260	-15	-15	-13	-27
	total casts that may fail	**1066**	**-137**	**-97**	**-94**	**-78**
	application reachable casts	531	-2	-2	0	-2
	application casts that may fail	485	-11	-43	-32	-21
	average var-points-to	**88.08**	**22.3**	**15.8**	**8.2**	**7.3**
	average application var-points-to	102.3	29.7	26.1	8.5	8.0

Figure 5. Precision metrics for 2plain+1H and 2full+1H for a subset of the DaCapo benchmarks. The last two metrics ("average ...") are in absolute numbers, the rest are given relative to the *immediately preceding* column (*not* relative to the numbers in the "insensitive" column). All metrics are end-user (i.e., context-insensitive) metrics. Var-points-to is the main relation of a points-to analysis, linking a variable to the allocation sites it may be referring to. (The average is over variables.) "Reachable methods" is the same as call-graph nodes, hence the first two metrics show how precise is the on-the-fly inferred call-graph. "Polymorphic call-sites" are those for which the analysis cannot statically determine a unique receiver method. "Casts that may fail" are those for which the analysis cannot statically determine that they are safe.

may fail at run-time (i.e., for which the analysis cannot statically determine that the cast is always safe). These metrics paint a fairly complete picture of relative analysis precision, although the rest of the metrics are useful for framing (e.g., for examining how the statistics vary between application classes and system libraries, for comparing to the total number of reachable call sites, etc.).

As can be seen in Figure 5, 2full+1H is almost always significantly more precise than 2plain+1H, even though both analyses have the same context depth. The difference in precision is quite substantial: For several metrics and programs (e.g., multiple metrics for pmd, or reduction in total polymorphic virtual call sites for many programs), the difference between full-object-sensitivity and plain-object-sensitivity is as large as any other single-step increment in precision (e.g., from 1-obj to 1obj+H).

Perhaps most impressively, this precision is accompanied by substantially improved performance. Figure 6 shows the running time of the analyses, together with two key internal complexity metrics: the number of edges in the context-sensitive callgraph (i.e., how many context-qualified methods are inferred to call how many other context-qualified methods) and the size of the context-sensitive var-points-to set, i.e., the total number of facts inferred that relate a context-qualified variable with a context-qualified allocation site.

The running time of 2full+1H is almost always much lower than that of 2plain+1H. The case of "chart" is most striking, with 2full+1H finishing in well under a third of the time of 2plain+1H, while achieving the much higher precision shown in Figure 5. The internal metrics show that 2full+1H makes excellent use of its context and has substantially lower internal complexity than 2plain+1H. Note that the statistics for context-sensitive var-points-to are quite low, explaining why the analysis is faster, since each variable needs to be examined only in a smaller number of contexts. A second reason why such internal metrics are important is that the performance of an analysis depends very much on algorithmic and data structure implementation choices, such as whether BDDs are used to represent large relations. Internal metrics, on the other hand, are invariant and indicate a complexity of the analysis that often transcends representation choices.

It is easy to see from the above figures that full-object-sensitivity makes a much better choice of context than plain-object-sensitivity, resulting in both increased precision and better performance. In fact, we have found the 2full+1H analysis to be a sweet spot in the current set of near-feasible analyses *in terms of precision*. Adding an extra level of context sensitivity for object fields, yielding a 2full+2H analysis, adds extremely little precision to the analysis results while greatly increasing the analysis cost.

In fact, for the benchmarks shown, 2full+1H is even significantly faster than the much less precise 1obj+H. Nevertheless, the same is not true universally. Of the 10 DaCapo benchmarks in our evaluation set, 2full+1H handles 6 with ease (the 5 shown plus "lusearch" which has very similar behavior to "luindex") but its running time explodes for the other 4. The 2plain+1H analysis explodes at least as badly for the 4 benchmarks, but a 1obj+H analysis handles 9 out of 10, and a 1-obj analysis handles all of them. In short, the performance (and internal complexity) of a highly-precise but deep-context analysis is bimodal: when precision is maintained, the analysis performs admirably. When, however, significant imprecision creeps in, the analysis does badly, since the number of contexts increases in combinatorial fashion.

Therefore, 2full+1H achieves excellent precision but is not a good point in the design space in terms of scalability. (In the past, the only fully scalable uses of object-sensitivity with depth > 1 have applied deep context to a small, carefully selected subset of allocation sites; we are interested in scalability when the whole program is analyzed with the precision of deep context.) This is the shortcoming that we expect to address with type-sensitive analyses.

5.3 Importance of Type Context Choice

In Section 4.2 we argued that, for judicious use of the extra context element, function $\mathcal{T}$ has to be defined so that it returns the enclosing type of an allocation site and not the type that is being allocated. Experimentally, this is very clearly the case. Figure 7 demonstrates this for two of our benchmark programs (the alphabetically first and last, which are representative of the rest). 1obj+H is shown as a baseline, to appreciate the difference. With the "wrong" type context, a 1type1obj+1H analysis is far more expensive and barely more precise than 1obj+H, while with the right type context the analysis is impressively scalable and precise (very close to 2full+1H, as we show later).

As can be seen, the impact of a good context is highly significant, both for scalability (in terms of time and internal metrics) and for precision. In our subsequent discussion we assume that all type-sensitive analyses use the superior context, as defined above.

5.4 Type-Sensitivity Precision and Performance

We found that type-sensitivity fully meets its stated goal: it yields analyses that are almost as precise as full-object-sensitive ones, while being highly scalable. In fact, type-sensitive analyses seem to clearly supplant other current state-of-the-art analyses—e.g., both 2type+1H and 1type1obj+1H seem overwhelmingly better than 1obj+H in both precision and performance for most of our benchmarks and metrics.

Our experiment space consists of the four precise analyses that appear feasible or mostly-feasible with current capabilities: 1obj+H, 2type+1H, 1type1obj+1H, and 2full+1H. Figure 8 shows the results of our evaluation for 8 of the 10 benchmark programs. (There is some replication of numbers compared to the previous tables, but this is limited to columns included as baselines.) We omit lusearch for layout reasons, since it behaves almost identically to luindex. We discuss the final benchmark, hsqldb, in text, but do not list it on the table because 2type+1H is the only of the four analyses that terminates on it.

Note that the first two analyses (1obj+H, 2type+1H) are semantically incomparable in precision but every other pair has a provably more precise analysis, so the issue concerns the amount of extra precision obtained and the running time cost. Specifically, 2full+1H is guaranteed to be more precise than the other three analyses, and 1type1obj+1H is guaranteed to be more precise than either 1obj+H or 2type+1H.

The trends from our experiments are quite clear:

- Although there is no guarantee, 2type+1H is almost always more precise than 1obj+H, hence the 2type+1H precision metrics (reported in the table relative to the preceding column, i.e., 1obj+H) are overwhelmingly showing negative numbers (i.e., an improvement). Additionally, 2type+1H is almost always (for 9 out of 10 programs) the fastest analysis in our set. In all but one case, 2type+1H is several times faster than 1obj+H—e.g., 5x faster or more for 4 of the benchmarks. The clear improvement of 2type+1H over 1obj+H is perhaps the most important of our experimental findings. Recall that 1obj+H is currently considered the "sweet spot" of precision and scalability in practice: a highly precise analysis, that is still feasible for large programs without exploding badly in complexity.
- 2type+1H achieves great scalability for fairly good precision. It is the only analysis that terminates for all our benchmark programs. It typically produces quite tight points-to sets, with antlr being a significant exception that requires more examination. In terms of client analyses and end-user metrics, the increase in pre-

		insensitive	1obj	1obj+H	2plain+1H	2full+1H
antlr	time (sec)	86.5	134.0	427.4	236.9	161.1
	context-sensitive callgraph edges (thousands)		1,484	966	1,428	2,458
	context-sensitive var-points-to (thousands)	13,143	8,147	49,237	24,980	9,279
chart	time (sec)	72.2	380.2	1199.2	**2496.0**	**688.2**
	context-sensitive callgraph edges (thousands)		1,463	1,087	9,564	7,469
	context-sensitive var-points-to (thousands)	7,054	19,942	83,354	107,221	22,854
eclipse	time (sec)	67.2	228.0	826.0	502.0	480.4
	context-sensitive callgraph edges (thousands)		1,921	1,278	2,103	5,341
	context-sensitive var-points-to (thousands)	5,754	9,962	64,586	65,435	22,574
luindex	time (sec)	37.9	63.2	179.3	123.9	124.3
	context-sensitive callgraph edges (thousands)		384	324	779	1,227
	context-sensitive var-points-to (thousands)	2,737	2,781	16,968	9,576	5,072
pmd	time (sec)	57.7	120.0	293.7	**392.6**	**160.0**
	context-sensitive callgraph edges (thousands)		553	418	3,610	1,614
	context-sensitive var-points-to (thousands)	4,392	5,314	24,902	35,628	6,770

Figure 6. Performance and complexity metrics for object-sensitive analyses. 2full+1H is almost always faster than 2plain+1H (some striking cases are highlighted). Additionally, 2full+1H makes good use of context and has often substantially lower internal metrics than 2plain+1H, and typically even than 1obj+H.

		1obj+H	1type1obj+1H	
			bad context	good context
antlr	call-graph edges	41280	-329	-1124
	reachable meths	**5692**	**-3**	**-78**
	reachable v-calls	27599	-2	-404
	poly v-calls	**1266**	**-51**	**-27**
	reach. v-calls in app	16393	0	-9
	poly v-calls in app	851	0	0
	reachable casts	1009	-1	-38
	casts that may fail	**614**	**-4**	**-157**
	reach. casts in app	308	0	-1
	casts in app may fail	216	0	-61
	avg var-points-to	**15.14**	**10.62**	**8.19**
	avg app var-points-to	15.25	9.02	8.51
	time (sec)	**427.4**	**376.7**	**114.2**
	c-s callgraph edge (K)	965	816	960
	c-s var-points-to (K)	49237	43030	7459

		1obj+H	1type1obj+1H	
			bad context	good context
xalan	call-graph edges	35908	-408	-1290
	reachable meths	**7237**	**-2**	**-86**
	reachable v-calls	19828	-2	-389
	poly v-calls	**1175**	**-52**	**-51**
	reach. v-calls in app	7709	0	0
	poly v-calls in app	726	-2	-6
	reachable casts	1264	-1	-37
	casts that may fail	**668**	**-5**	**-123**
	reach. casts in app	501	0	0
	casts in app may fail	250	-4	-23
	avg var-points-to	**14.94**	**14.03**	**9.57**
	avg app var-points-to	15.73	15.14	11.58
	time (sec)	**979.9**	**4398.9**	**831.0**
	c-s callgraph edge (K)	936	4915	2580
	c-s var-points-to (K)	96021	163916	38205

Figure 7. Precision, performance, and internal complexity metrics for a type-object-sensitive analysis with a good and a bad choice of context. The entries are the same as in Figures 5 and 6, with metric names condensed. As in Figure 5, all but the last two precision metrics are reported as differences relative to the *immediately preceding* column (i.e., we are showing how much more precision the good context yields over the already higher precision of the bad context, not over the baseline).

cision going from 1obj+H to 2type+1H is often greater than that of going from 2type+1H to 2full+1H. Overall, 2type+1H is an excellent approximation of 2full+1H given its low cost.

- Although not shown on the table, 2type+1H is not just faster than the three shown analyses but also faster than 1obj, for 7 out of 10 benchmark programs. The difference in precision between the two analyses is enormous, however. A good example is the hsqldb benchmark, omitted from Figure 8 since 2type+1H is the only analysis with a context-sensitive heap that terminates for it. 2type+1H processes hsqldb slightly faster than 1obj (404sec instead of 464sec). At the same time, all precision metrics are drastically better. The points-to sets are almost half the size (11.3 total vs. 22.1, and 13.2 for application vars only vs. 18.2). On other precision metrics the difference between 2type+1H and 1obj is much greater than that between 1obj and a context-insensitive analysis. For "application casts that may fail" alone, 2type+1obj eliminates 117 instances relative to 1obj.
- 1type1obj+1H is highly precise and its difference from 2full+1H is rarely significant. (It is illuminating to add the two columns and compare the cumulative difference of 1type1obj+1H from 1obj+1H, relative to the difference of the former from 2full+1H.) At the same time, 1type1obj+1H avoids many of the scalability problems of 2full+1H: it terminates on 8 of 10 benchmarks (instead of 6 out of 10) and is always faster than 2full+1H, occasionally (e.g., chart) by a significant factor.

6. Conclusions

In this paper we strove for a better understanding of the concept of object-sensitivity in points-to analysis. Our exploration led to a precise formal modeling, to a complete mapping of past object-sensitive analyses in the literature, as well as to insights on how context affects the precision and scalability of an analysis. One concrete outcome of our work is to establish full-object-sensitivity (and especially a 2full+1H analysis) as a superior choice of context compared to others in past literature. Additionally, we have introduced the concept of type-sensitivity and applied our insights to pick an appropriate type to use as context of a points-to analysis. The result is a range of analyses, especially 2type+1H and 1type1obj+1H, that have very good to excellent scalability, while maintaining most of the precision of a much more expensive analysis. The new analyses we introduced are current sweet spots in the design space and represent a significant advancement of the state-of-the-art in points-to analysis.

		1obj+H	2type +1H	1type 1obj+1H	2full +1H
antlr	call-graph edges	41280	-1401	-52	-44
	reachable meths	**5692**	**-77**	**-4**	**-2**
	reachable v-calls	27599	-405	-1	-5
	poly v-calls	**1266**	**-70**	**-8**	**-28**
	reach.v-calls in app	16393	-9	0	0
	poly v-calls in app	851	0	0	0
	reachable casts	1009	-39	0	0
	casts that may fail	**614**	**-104**	**-57**	**-47**
	reach. casts in app	308	-1	0	0
	app casts may fail	216	-53	-8	-28
	avg var-points-to	**15.1**	**23.0**	**8.2**	**8.2**
	avg app v-points-to	15.3	41.7	8.5	8.5
	time (sec)	**427.4**	**78.8**	**114.2**	**161.1**
	c-s callgraph edge (K)	966	512	960	2,458
	c-s var-points-to (K)	49,237	4,029	7,459	9,279
bloat	call-graph edges	47792	-1797	-489	
	reachable meths	**6945**	**-85**	**-12**	
	reachable v-calls	25220	-404	-18	
	poly v-calls	**1406**	**-82**	**-85**	
	reach.v-calls in app	13879	-20	0	
	poly v-calls in app	953	-28	-58	
	reachable casts	2062	-43	-2	
	casts that may fail	**1546**	**-45**	**-120**	
	reach. casts in app	1346	-2	0	
	casts in app may fail	1134	13	-70	
	avg var-points-to	**32.5**	**21.8**	**18.6**	
	avg app var-points-to	42.6	31.9	29.1	
	time (sec)	**2307.2**	**432.7**	**2431.0**	
	c-s callgraph edge (K)	1,791	1,036	3,196	
	c-s var-points-to (K)	73,527	10,375	43,073	
chart	call-graph edges	41628	-2776	-191	-85
	reachable meths	**8339**	**-133**	**-27**	**-8**
	reachable v-calls	23384	-491	-39	-10
	poly v-calls	**1104**	**-155**	**-20**	**-27**
	reach. v-calls in app	3641	-30	-29	0
	poly v-calls in app	93	-8	-6	0
	reachable casts	1668	-55	-10	0
	casts that may fail	**1023**	**-39**	**-199**	**-46**
	reach. casts in app	228	-15	-7	0
	casts in app may fail	115	3	-66	-7
	avg var-points-to	**20.1**	**8.5**	**6.8**	**6.7**
	avg app var-points-to	14.4	4.0	2.8	2.8
	time (sec)	**1199.2**	**143.2**	**199.0**	**688.2**
	c-s callgraph edge (K)	1,087	974	1,252	7,469
	c-s var-points-to (K)	83,354	6,572	9,093	22,854
eclipse	call-graph edges	32216	-2575	-86	-42
	reachable meths	**6321**	**-127**	**-4**	**-1**
	reachable v-calls	17841	-577	-1	-4
	poly v-calls	**752**	**-77**	**-24**	**-26**
	reach. v-calls in app	5805	-74	-1	0
	poly v-calls in app	243	3	-7	-2
	reachable casts	1228	-51	0	0
	casts that may fail	**685**	**-58**	**-98**	**-28**
	reach. casts in app	471	-8	0	0
	casts in app may fail	246	-5	-44	-8
	avg var-points-to	**16.1**	**11.8**	**9.1**	**9.1**
	avg app var-points-to	15.8	13.0	9.5	9.4
	time (sec)	**826.0**	**178.0**	**301.1**	**480.4**
	c-s callgraph edge (K)	1,278	1,314	2,241	5,341
	c-s var-points-to (K)	64,586	9,232	17,154	22,574

		1obj+H	2type +1H	1type 1obj+1H	2full +1H
jython	call-graph edges	30370	-2091		
	reachable meths	**5754**	**-118**		
	reachable v-calls	16057	-830		
	poly v-calls	**768**	**-71**		
	reach. v-calls in app	7146	-492		
	poly v-calls in app	422	0		
	reachable casts	1272	-18		
	casts that may fail	**741**	**-11**		
	reach. casts in app	677	0		
	casts in app may fail	445	17		
	avg var-points-to	**21.2**	**19.1**		
	avg app var-points-to	30.7	31.4		
	time (sec)	**1215.7**	**2107.6**		
	c-s callgraph edge (K)	923	4,399		
	c-s var-points-to (K)	110,113	53,552		
luindex	call-graph edges	22251	-1368	-63	-42
	reachable meths	**4677**	**-78**	**-4**	**-2**
	reachable v-calls	12365	-400	-1	-5
	poly v-calls	**435**	**-66**	**-12**	**-26**
	reach. v-calls in app	1173	-4	0	0
	poly v-calls in app	29	4	-4	0
	reachable casts	761	-38	0	0
	casts that may fail	**429**	**-54**	**-66**	**-19**
	reach. casts in app	62	0	0	0
	casts in app may fail	25	3	-18	0
	avg var-points-to	**10.8**	**7.8**	**6.4**	**6.3**
	avg app var-points-to	5.9	4.2	2.9	2.9
	time (sec)	**179.3**	**67.7**	**80.8**	**124.3**
	c-s callgraph edge (K)	324	473	656	1,227
	c-s var-points-to (K)	16,968	2,848	3,892	5,072
pmd	call-graph edges	29134	-1720	-52	-80
	reachable meths	**6080**	**-86**	**-5**	**-8**
	reachable v-calls	15722	-475	-1	-12
	poly v-calls	**508**	**-83**	**-8**	**-37**
	reach. v-calls in app	4436	-84	0	-4
	poly v-calls in app	81	-17	0	-11
	reachable casts	1230	-40	0	0
	casts that may fail	**832**	**-63**	**-75**	**-34**
	reach. casts in app	527	-2	0	0
	casts in app may fail	431	-12	-26	-15
	avg var-points-to	**15.8**	**8.8**	**7.3**	**7.3**
	avg app var-points-to	26.1	8.9	8.0	8.0
	time (sec)	**293.7**	**78.2**	**128.1**	**160.0**
	c-s callgraph edge (K)	418	527	1,043	1,614
	c-s var-points-to (K)	24,902	3,370	5,901	6,770
xalan	call-graph edges	35908	-1553	-145	
	reachable meths	**7237**	**-65**	**-23**	
	reachable v-calls	19828	-355	-36	
	poly v-calls	**1175**	**-88**	**-15**	
	reach. v-calls in app	7709	35	-35	
	poly v-calls in app	726	-1	-7	
	reachable casts	1264	-34	-4	
	casts that may fail	**668**	**21**	**-149**	
	reach. casts in app	501	4	-4	
	casts in app may fail	250	52	-79	
	avg var-points-to	**14.9**	**12.1**	**9.6**	
	avg app var-points-to	15.7	14.4	11.6	
	time (sec)	**979.9**	**435.8**	**831.0**	
	c-s callgraph edge (K)	936	2,871	2,580	
	c-s var-points-to (K)	96,021	21,717	38,205	

Figure 8. Precision, performance, and internal complexity metrics for precise analyses. The entries are the same as in Figures 5 and 6, with metric names condensed. As in Figure 5, all but the last two precision metrics are reported as differences relative to the *immediately preceding* column. Empty entries are due to non-termination after 2 hours of running time. Some of the most interesting metrics are highlighted.

Acknowledgments

We thank the anonymous reviewers for their helpful comments, as well as Matt Might and David Van Horn for interesting discussions. This work was funded by the National Science Foundation under grants CCF-0917774 and CCF-0934631, as well as by the Natural Sciences and Engineering Research Council of Canada.

References

[1] Ole Agesen. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, pages 2–26, London, UK, 1995. Springer-Verlag.

[2] Martin Bravenboer and Yannis Smaragdakis. Exception analysis and points-to analysis: Better together. In Laura Dillon, editor, *ISSTA '09: Proceedings of the 2009 International Symposium on Software Testing and Analysis*, New York, NY, USA, July 2009.

[3] Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *OOPSLA '09: 24th annual ACM SIGPLAN conference on Object Oriented Programming, Systems, Languages, and Applications*, New York, NY, USA, 2009. ACM.

[4] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[5] Stephen Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. In *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis*, pages 133–144, New York, NY, USA, 2006. ACM.

[6] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001.

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

[8] Ondřej Lhoták and Laurie 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.

[9] Ondřej Lhoták and Laurie Hendren. Relations as an abstraction for BDD-based program analysis. *ACM Trans. Program. Lang. Syst.*, 30(4):1–63, 2008.

[10] Donglin Liang, Maikel Pennings, and Mary Jean Harrold. Evaluating the impact of context-sensitivity on Andersen's algorithm for Java programs. In Michael D. Ernst and Thomas P. Jensen, editors, *PASTE*, pages 6–12. ACM, 2005.

[11] Benjamin Livshits, John Whaley, and Monica S. Lam. Reflection analysis for Java. In Kwangkeun Yi, editor, *Proceedings of the 3rd Asian Symposium on Programming Languages and Systems*, volume 3780. Springer-Verlag, November 2005.

[12] Matthew Might, Yannis Smaragdakis, and David Van Horn. Resolving and exploiting the k-CFA paradox: Illuminating functional vs. object-oriented program analysis. In *Conf. on Programming Language Design and Implementation (PLDI)*. ACM, June 2010.

[13] 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.

[14] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for Java. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'06)*, pages 308–319, 2006.

[15] John Plevyak and Andrew A. Chien. Precise concrete type inference for object-oriented languages. In *Proceedings of the ninth annual conference on Object-oriented programming systems, language, and applications*, OOPSLA '94, pages 324–340, New York, NY, USA, 1994. ACM.

[16] John Reppy. Type-sensitive control-flow analysis. In *Proceedings of the 2006 ACM SIGPLAN Workshop on ML*, pages 74–83, September 2006.

[17] Barbara G. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. In Görel Hedin, editor, *Compiler Construction, 12th International Conference, CC 2003*, volume 2622 of *Lecture Notes in Computer Science*, pages 126–137. Springer, 2003.

[18] Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis*, pages 189–233, Englewood Cliffs, NJ, 1981. Prentice-Hall, Inc.

[19] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, May 1991.

[20] Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for Java. In *PLDI '06: Proc. of the 2006 ACM SIGPLAN conf. on Programming language design and implementation*, pages 387–400, New York, NY, USA, 2006. ACM.

[21] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for Java. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 264–280. ACM Press, 2000.

[22] John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. Using Datalog with binary decision diagrams for program analysis. In Kwangkeun Yi, editor, *APLAS*, volume 3780 of *Lecture Notes in Computer Science*, pages 97–118. Springer, 2005.

[23] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI '04: Proc. of the ACM SIGPLAN 2004 conf. on Programming language design and implementation*, pages 131–144, New York, NY, USA, 2004. ACM.