Interprocedural Analysis with Lazy Propagation

Simon Holm Jensen[1,⋆], Anders Møller[1,⋆,⋆⋆], and Peter Thiemann[2]

[1] Aarhus University, Denmark
{simonhj,amoeller}@cs.au.dk
[2] Universität Freiburg, Germany
thiemann@informatik.uni-freiburg.de

Abstract. We propose *lazy propagation* as a technique for flow- and context-sensitive interprocedural analysis of programs with objects and first-class functions where transfer functions may not be distributive. The technique is described formally as a systematic modification of a variant of the monotone framework and its theoretical properties are shown. It is implemented in a type analysis tool for JavaScript where it results in a significant improvement in performance.

1 Introduction

With the increasing use of object-oriented scripting languages, such as JavaScript, program analysis techniques are being developed as an aid to the programmers [7, 8, 29, 27, 2, 9]. Although programs written in such languages are often relatively small compared to typical programs in other languages, their highly dynamic nature poses difficulties to static analysis. In particular, JavaScript programs involve complex interplays between first-class functions, objects with modifiable prototype chains, and implicit type coercions that all must be carefully modeled to ensure sufficient precision.

While developing a program analysis for JavaScript [14] aiming to statically infer type information we encountered the following challenge: *How can we obtain a flow- and context-sensitive interprocedural dataflow analysis that accounts for mutable heap structures, supports objects and first-class functions, is amenable to non-distributive transfer functions, and is efficient and precise?* Various directions can be considered. First, one may attempt to apply the classical monotone framework [18] as a whole-program analysis with an iterative fixpoint algorithm, where function call and return flow is treated as any other dataflow. This approach turns out to be unacceptable: the fixpoint algorithm requires too many iterations, and precision may suffer because spurious dataflow appears via interprocedurally unrealizable paths. Another approach is to apply the IFDS technique [23], which eliminates those problems. However, it is restricted to distributive analyses, which makes it inapplicable in our situation. A further

⋆ Supported by The Danish Research Council for Technology and Production, grant no. 274-07-0488.

⋆⋆ Corresponding author.

R. Cousot and M. Martel (Eds.): SAS 2010, LNCS 6337, pp. 320–339, 2010.

consideration is the functional approach [26] which models each function in the program as a partial summary function that maps input dataflow facts to output dataflow facts and then uses this summary function whenever the function is called. However, with a dataflow lattice as large as in our case it becomes difficult to avoid reanalyzing each function a large number of times. Although there are numerous alternatives and variations of these approaches, we have been unable to find one in the literature that adequately addresses the challenge described above. Much effort has also been put into more specialized analyses, such as pointer analysis [10], however it is far from obvious how to generalize that work to our setting.

As an introductory example, consider this fragment of a JavaScript program:

```
function Person(n) { this.setName(n); }
Person.prototype.setName = function(n) { this.name = n; }
function Student(n,s) { Person.call(this, n);
                        this.studentid = s.toString(); }
Student.prototype = new Person;
var x = new Student("John Doe", 12345);
x.setName("John Q. Doe");
```

The code defines two "classes" with constructors `Person` and `Student`. `Person` has a method `setName` via its prototype object, and `Student` inherits `setName` and defines an additional field `studentid`. The `call` statement in `Student` invokes the super class constructor `Person`.

Analyzing the often intricate flow of control and data in such programs requires detailed modeling of points-to relations among objects and functions and of type coercion rules. TAJS is a whole-program analysis based on the monotone framework that follows this approach, and our first implementation is capable of analyzing complex properties of many JavaScript programs. However, our experiments have shown a considerable redundancy of computation during the analysis that causes simple functions to be analyzed a large number of times. If, for example, the `setName` method is called from other locations in the program, then the slightest change of any abstract state appearing at any call site of `setName` during the analysis would cause the method to be reanalyzed, even though the changes may be entirely irrelevant for that method. In this paper, we propose a technique for avoiding much of this redundancy while preserving, or even improving, the precision of the analysis. Although our main application is type analysis for JavaScript, we believe the technique is more generally applicable to analyses for object-oriented languages.

The main idea is to introduce a notion of "unknown" values for object fields that are not accessed within the current function. This prevents much irrelevant information from being propagated during the fixpoint computation. The analysis initially assumes that no fields are accessed when flow enters a function. When such an unknown value is read, a recovery operation is invoked to go back through the call graph and propagate the correct value. By avoiding to recover the same values repeatedly, the total amortized cost of recovery is never higher

than that of the original analysis. With large abstract states, the mechanism makes a noticeable difference to the analysis performance.

Lazy propagation should not be confused with demand-driven analysis [13]. The goal of the latter is to compute the results of an analysis only at specific program points thereby avoiding the effort to compute a global result. In contrast, lazy propagation computes a model of the state for each program point.

The contributions of this paper can be summarized as follows:

- We propose an ADT-based adaptation of the monotone framework to programming languages with mutable heap structures and first-class functions and exhibit some of its limitations regarding precision and performance.
- We describe a systematic modification of the framework that introduces *lazy propagation*. This novel technique propagates dataflow facts "by need" in an iterative fixpoint algorithm. We provide a formal description of the method to reason about its properties and to serve as a blueprint for an implementation.
- The lazy propagation technique is experimentally validated: It has been implemented into our type analysis for JavaScript, TAJS [14], resulting in a significant improvement in performance.

In the full version of the paper [15], we also prove termination, relate lazy propagation with the basic framework—showing that precision does not decrease, and sketch a soundness proof of the analysis.

2 A Basic Analysis Framework

Our starting point is the classical monotone framework [18] tailored to programming languages with mutable heap structures and first-class functions. The mutable state consists of a heap of objects. Each object is a map from field names to values, and each value is either a reference to an object, a function, or some primitive value. Note that this section contains no new results, but it sets the stage for presenting our approach in Section 3.

2.1 Analysis Instances

Given a program Q, an instance of the monotone framework for an analysis of Q is a tuple $\mathcal{A} = (F, N, L, P, C, n_0, c_0, \mathsf{Base}, T)$ consisting of:

F: the set of *functions* in Q;
N: the set of *primitive statements* (also called *nodes*) in Q;
L: a set of *object labels* in Q;
P: a set of *field names* (also called *properties*) in Q;
C: a set of abstract *contexts*, which are used for context sensitivity;
$n_0 \in N$ and $c_0 \in C$: an initial statement and context describing the entry of Q;
Base: a *base lattice* for modeling primitive values, such as integers or booleans;

$T : C \times N \to \mathsf{AnalysisLattice} \to \mathsf{AnalysisLattice}$: a monotone *transfer function* for each primitive statement, where AnalysisLattice is a lattice derived from the above information as detailed in Section 2.2.

Each of the sets must be finite and the Base lattice must have finite height. The primitive statements are organized into intraprocedural control flow graphs [19], and the set of object labels is typically determined by allocation-site abstraction [16, 5].

The notation $\mathit{fun}(n) \in F$ denotes the function that contains the statement $n \in N$, and $\mathit{entry}(f)$ and $\mathit{exit}(f)$ denote the unique entry statement and exit statement, respectively, of the function $f \in F$. For a function call statement $n \in N$, $\mathit{after}(n)$ denotes the statement being returned to after the call. A *location* is a pair (c, n) of a context $c \in C$ and a statement $n \in N$.

2.2 Derived Lattices

An analysis instance gives rise to a collection of derived lattices. In the following, each function space is ordered pointwise and each powerset is ordered by inclusion. For a lattice X, the symbols $\bot_X$, $\sqsubseteq_X$, and $\sqcup_X$ denote the bottom element (representing the absence of information), the partial order, and the least upper bound operator (for merging information). We omit the X subscript when it is clear from the context.

An *abstract value* is described by the lattice Value as a set of object labels, a set of functions, and an element from the base lattice:

$$\mathsf{Value} = \mathcal{P}(L) \times \mathcal{P}(F) \times \mathsf{Base}$$

An *abstract object* is a map from field names to abstract values:

$$\mathsf{Obj} = P \to \mathsf{Value}$$

An *abstract state* is a map from object labels to abstract objects:

$$\mathsf{State} = L \to \mathsf{Obj}$$

Call graphs are described by this powerset lattice:

$$\mathsf{CallGraph} = \mathcal{P}(C \times N \times C \times F)$$

In a call graph $g \in \mathsf{CallGraph}$, we interpret $(c_1, n_1, c_2, f_2) \in g$ as a potential function call from statement n_1 in context c_1 to function f_2 in context c_2.

Finally, an element of AnalysisLattice provides an abstract state for each context and primitive statement (in a forward analysis, the program point immediately *before* the statement), combined with a call graph:

$$\mathsf{AnalysisLattice} = (C \times N \ \to \ \mathsf{State}) \times \mathsf{CallGraph}$$

In practice, an analysis may involve additional lattice components such as an abstract stack or extra information associated with each abstract object or field. We omit such components to simplify the presentation as they are irrelevant to the features that we focus on here. Our previous paper [14] describes the full lattices used in our type analysis for JavaScript.

$solve(\mathcal{A})$ **where** $\mathcal{A} = (F, N, L, P, C, n_0, c_0, \mathsf{Base}, T)$:
  $a := \bot_{\mathsf{AnalysisLattice}}$
  $W := \{(c_0, n_0)\}$
  while $W \neq \emptyset$ **do**
    select and remove (c, n) from W
    $T_a(c, n)$
  end while
  return a

Fig. 1. The worklist algorithm. The worklist contains *locations*, i.e., pairs of a context and a statement. The operation $T_a(c, n)$ computes the transfer function for (c, n) on the current analysis lattice element a and updates a accordingly. Additionally, it may add new entries to the worklist W. The transfer function for the initial location (c_0, n_0) is responsible for creating the initial abstract state.

2.3 Computing the Solution

The *solution* to $\mathcal{A}$ is the least element $a \in \mathsf{AnalysisLattice}$ that solves these constraints:

$$\forall c \in C, n \in N : T(c, n)(a) \sqsubseteq a$$

Computing the solution to the constraints involves fixpoint iteration of the transfer functions, which is typically implemented with a worklist algorithm as the one presented in Figure 1. The algorithm maintains a worklist $W \subseteq C \times N$ of locations where the abstract state has changed and thus the transfer function should be applied. Lattice elements representing functions, in particular $a \in \mathsf{AnalysisLattice}$, are generally considered as *mutable* and we use the notation $T_a(c, n)$ for the assignment $a := T(c, n)(a)$. As a side effect, the call to $T_a(c, n)$ is responsible for adding entries to the worklist W, as explained in Section 2.4. This slightly unconventional approach to describing fixpoint iteration simplifies the presentation in the subsequent sections.

Note that the solution consists of both the computed call graph and an abstract state for each location. We do not construct the call graph in a preliminary phase because the presence of first-class functions implies that dataflow facts and call graph information are mutually dependent (as evident from the example program in Section 1).

This fixpoint algorithm leaves two implementation choices: the order in which entries are removed from the worklist W, which can greatly affect the number of iterations needed to reach the fixpoint, and the representation of lattice elements, which can affect both time and memory usage. These choices are, however, not the focus of the present paper (see, e.g. [17, 19, 12, 3, 28]).

2.4 An Abstract Data Type for Transfer Functions

To precisely explain our modifications of the framework in the subsequent sections, we treat AnalysisLattice as an imperative ADT (abstract data type) [20] with the following operations:

- *getfield* : $C \times N \times L \times P \to \mathsf{Value}$
- *getcallgraph* : $() \to \mathsf{CallGraph}$
- *getstate* : $C \times N \to \mathsf{State}$
- *propagate* : $C \times N \times \mathsf{State} \to ()$
- *funentry* : $C \times N \times C \times F \times \mathsf{State} \to ()$
- *funexit* : $C \times N \times C \times F \times \mathsf{State} \to ()$

We let $a \in \mathsf{AnalysisLattice}$ denote the current, mutable analysis lattice element. The transfer functions can only access a through these operations.

The operation $\mathit{getfield}(c, n, \ell, p)$ returns the abstract value of the field p in the abstract object ℓ at the entry of the primitive statement n in context c. In the basic framework, *getfield* performs a simple lookup, without any side effects on the analysis lattice element:

$a.\mathit{getfield}(c \in C, n \in N, \ell \in L, p \in P)$:
return $u(\ell)(p)$ **where** $(m, \_) = a$ **and** $u = m(c, n)$

The *getcallgraph* operation selects the call graph component of the analysis lattice element:

$a.\mathit{getcallgraph}()$:
return g **where** $(\_, g) = a$

Transfer functions typically use the *getcallgraph* operation in combination with the *funexit* operation explained below. Moreover, the *getcallgraph* operation plays a role in the extended framework presented in Section 3.

The *getstate* operation returns the abstract state at a given location:

$a.\mathit{getstate}(c \in C,\ n \in N)$:
return $m(c, n)$ **where** $(m, \_) = a$

The transfer functions must not read the field values from the returned abstract state (for that, the *getfield* operation is to be used). They may construct parameters to the operations *propagate*, *funentry*, and *funexit* by updating a copy of the returned abstract state.

The transfer functions must use the operation $\mathit{propagate}(c, n, s)$ to pass information from one location to another within the same function (excluding recursive function calls). As a side effect, *propagate* adds the location (c, n) to the worklist W if its abstract state has changed. In the basic framework, *propagate* is defined as follows:

$a.\mathit{propagate}(c \in C,\ n \in N,\ s \in \mathsf{State})$:
let $(m, g) = a$
if $s \not\sqsubseteq m(c, n)$ **then**
$m(c, n) := m(c, n) \sqcup s$
$W := W \cup \{(c, n)\}$
end if

The operation $\mathit{funentry}(c_1, n_1, c_2, f_2, s)$ models function calls in a forward analysis. It modifies the analysis lattice element a to reflect the possibility of a function

call from a statement n_1 in context c_1 to a function entry statement $entry(f_2)$ in context c_2 where s is the abstract state after parameter passing. (With languages where parameters are passed via the stack, which we ignore here, the lattice is augmented accordingly.) In the basic framework, *funentry* adds the call edge from (c_1, n_1) to (c_2, f_2) and propagates s into the abstract state at the function entry statement $entry(f_2)$ in context c_2:

$a.funentry(c_1 \in C,\ n_1 \in N,\ c_2 \in C,\ f_2 \in F,\ s \in \mathsf{State})$:
  $g := g \cup \{(c_1, n_1, c_2, f_2)\}$ **where** $(\_, g) = a$
  $a.propagate(c_2, entry(f_2), s)$
  $a.funexit(c_1, n_1, c_2, f_2, m(c_2, exit(f_2)))$

Adding a new call edge also triggers a call to *funexit* to establish dataflow from the function exit to the successor of the new call site.

The operation $funexit(c_1, n_1, c_2, f_2, s)$ is used for modeling function returns. It modifies the analysis lattice element to reflect the dataflow of s from the exit of a function f_2 in callee context c_2 to the successor of the call statement n_1 with caller context c_1. The basic framework does so by propagating s into the abstract state at the latter location:

$a.funexit(c_1 \in C,\ n_1 \in N,\ c_2 \in C,\ f_2 \in F,\ s \in \mathsf{State})$:
  $a.propagate(c_1, after(n_1), s)$

The parameters c_2 and f_2 are not used in the basic framework; they will be used in Section 3. The transfer functions obtain the connections between callers and callees via the *getcallgraph* operation explained earlier. If using an augmented lattice where the call stack is also modeled, that component would naturally be handled differently by *funexit* simply by copying it from the call location (c_1, n_1), essentially as local variables are treated in, for example, IFDS [23].

This basic framework is sufficiently general as a foundation for many analyses for object-oriented programming languages, such as Java or C#, as well as for object-based scripting languages like JavaScript as explained in Section 4. At the same time, it is sufficiently simple to allow us to precisely demonstrate the problems we attack and our solution in the following sections.

2.5 Problems with the Basic Analysis Framework

The first implementation of TAJS, our program analysis for JavaScript, is based on the basic analysis framework. Our initial experiments showed, perhaps not surprisingly, that many simple functions in our benchmark programs were analyzed over and over again (even for the same calling contexts) until the fixpoint was reached.

For example, a function in the `richards.js` benchmark from the V8 collection was analyzed 18 times when new dataflow appeared at the function entry:

```
TaskControlBlock.prototype.markAsRunnable = function () {
  this.state = this.state | STATE_RUNNABLE;
};
```

Most of the time, the new dataflow had nothing to do with the `this` object or the `STATE_RUNNABLE` variable. Although this particular function body is very short, it still takes time and space to analyze it and similar situations were observed for more complex functions and in other benchmark programs.

In addition to this abundant redundancy, we observed – again not surprisingly – a significant amount of spurious dataflow resulting from interprocedurally invalid paths. For example, if the function above is called from two different locations, with the same calling context, their entire heap structures (that is, the State component in the lattice) become joined, thereby losing precision.

Another issue we noticed was time and space required for propagating the initial state, which consists of 161 objects in the case of JavaScript. These objects are mutable and the analysis must account for changes made to them by the program. Since the analysis is both flow- and context-sensitive, a typical element of AnalysisLattice carries a lot of information even for small programs.

Our first version of TAJS applied two techniques to address these issues: (1) Lattice elements were represented in memory using *copy-on-write* to make their constituents shared between different locations until modified. (2) The lattice was extended to incorporate a simple effect analysis called *maybe-modified*: For each object field, the analysis would keep track of whether the field might have been modified since entering the current function. At function exit, field values that were definitely not modified by the function would be replaced by the value from the call site. As a consequence, the flow of unmodified fields was not affected by function calls. Although these two techniques are quite effective, the lazy propagation approach that we introduce in the next section supersedes the maybe-modified technique and renders copy-on-write essentially superfluous. In Section 4 we experimentally compare lazy propagation with both the basic framework and the basic framework extended with the copy-on-write and maybe-modified techniques.

3 Extending the Framework with Lazy Propagation

To remedy the shortcomings of the basic framework, we propose an extension that can help reducing the observed redundancy and the amount of information being propagated by the transfer functions. The key idea is to ensure that the fixpoint solver *propagates information "by need"*. The extension consists of a systematic modification of the ADT representing the analysis lattice. This modification implicitly changes the behavior of the transfer functions without touching their implementation.

3.1 Modifications of the Analysis Lattice

In short, we modify the analysis lattice as follows:

1. We introduce an additional abstract value, unknown. Intuitively, a field p of an object has this value in an abstract state associated with some location in

a function f if the value of p is not known to be needed (that is, referenced) in f or in a function called from f.
2. Each call edge is augmented with an abstract state that captures the data flow along the edge after parameter passing, such that this information is readily available when resolving unknown field values.
3. A special abstract state, none, is added, for describing absent call edges and locations that may be unreachable from the program entry.

More formally, we modify three of the sub-lattices as follows:

$$\mathsf{Obj} = P \to \left(\mathsf{Value}\downarrow_{\mathsf{unknown}}\right)$$

$$\mathsf{CallGraph} = C \times N \times C \times F \to (\mathsf{State}\downarrow_{\mathsf{none}})$$

$$\mathsf{AnalysisLattice} = \left(C \times N \ \to \ (\mathsf{State}\downarrow_{\mathsf{none}})\right) \times \mathsf{CallGraph}$$

Here, $X\downarrow_{\mathsf{y}}$ means the lattice X lifted over a new bottom element y. In a call graph $g \in \mathsf{CallGraph}$ in the original lattice, the presence of an edge $(c_1, n_1, c_2, f_2) \in g$ is modeled by $g'(c_1, n_1, c_2, f_2) \neq \mathsf{none}$ for the corresponding call graph g' in the modified lattice. Notice that $\bot_{\mathsf{State}}$ is now the function that maps all object labels and field names to $\mathsf{unknown}$, which is different from the element none.

3.2 Modifications of the Abstract Data Type Operations

Before we describe the systematic modifications of the ADT operations we motivate the need for an auxiliary operation, *recover*, on the ADT:

$$\mathit{recover} : C \times N \times L \times P \to \mathsf{Value}$$

Suppose that, during the fixpoint iteration, a transfer function $T_a(c, n)$ invokes $a.\mathit{getfield}(c, n, \ell, p)$ with the result $\mathsf{unknown}$. This result indicates the situation that the field p of an abstract object ℓ is referenced at the location (c, n), but the field value has not yet been propagated to this location due to the lazy propagation. The *recover* operation can then compute the proper field value by performing a specialized fixpoint computation to propagate just that field value to (c, n). We explain in Section 3.3 how *recover* is defined.

The *getfield* operation is modified such that it invokes *recover* if the desired field value is $\mathsf{unknown}$, as shown in Figure 2. The modification may break monotonicity of the transfer functions, however, the analysis still produces the correct result [15].

Similarly, the *propagate* operation needs to be modified to account for the lattice element none and for the situation where $\mathsf{unknown}$ is joined with an ordinary element. The latter is accomplished by using *recover* whenever this situation occurs. The resulting operation $\mathit{propagate}'$ is shown in Figure 3.

We then modify $\mathit{funentry}(c_1, n_1, c_2, f_2, s)$ such that the abstract state s is propagated "lazily" into the abstract state at the primitive statement $\mathit{entry}(f_2)$ in context c_2. Here, laziness means that every field value that, according to a, is not referenced within the function f_2 in context c_2 gets replaced by $\mathsf{unknown}$ in the abstract state. Additionally, the modified operation records the abstract

```
a.getfield′(c ∈ C, n ∈ N, ℓ ∈ L, p ∈ P):
  if m(c,n) ≠ none where (m, _) = a then
    v := a.getfield(c,n,ℓ,p)
    if v = unknown then
      v := a.recover(c,n,ℓ,p)
    end if
    return v
  else
    return ⊥Value
  end if
```

Fig. 2. Algorithm for $getfield'(c, n, \ell, p)$. This modified version of *getfield* invokes *recover* in case the desired field value is unknown. If the state is none according to a, the operation simply returns $\bot_{\mathsf{Value}}$.

```
a.propagate′(c ∈ C, n ∈ N, s ∈ State):
  let (m,g) = a and u = m(c,n)
  s′ := s
  if u ≠ none then
    for all ℓ ∈ L, p ∈ P do
      if u(ℓ)(p) = unknown ∧ s(ℓ)(p) ≠ unknown then
        u(ℓ)(p) := a.recover(c,n,ℓ,p)
      else if u(ℓ)(p) ≠ unknown ∧ s(ℓ)(p) = unknown then
        s′(ℓ)(p) := a.recover(c,n,ℓ,p)
      end if
    end for
  end if
  a.propagate(c,n,s′)
```

Fig. 3. Algorithm for $propagate'(c, n, s)$. This modified version of *propagate* takes into account that field values may be unknown in both a and s. Specifically, it uses *recover* to ensure that the invocation of *propagate* in the last line never computes the least upper bound of unknown and an ordinary field value. The treatment of unknown values in s assumes that s is recoverable with respect to the current location (c, n). If the abstract state at (c, n) is none (the least element), then that gets updated to s.

state at the call edge as required in the modified CallGraph lattice. The resulting operation $funentry'$ is defined in Figure 4. (Without loss of generality, we assume that the statement at $exit(f_2)$ returns to the caller without modifying the state.) As consequence of the modification, unknown field values get introduced into the abstract states at function entries.

The *funexit* operation is modified such that every unknown field value appearing in the abstract state being returned is replaced by the corresponding field value from the call edge, as shown in Figure 5. In JavaScript, entering a function body at a functions call affects the heap, which is the reason for using the state from the call edge rather than the state from the call statement. If we extended the lattice to also model the call stack, then that component would naturally be recovered from the call statement rather than the call edge.

```
a.funentry'(c1 ∈ C, n1 ∈ N, c2 ∈ C, f2 ∈ F, s ∈ State):
  let (m, g) = a and u = m(c2, entry(f2))
  // update the call edge
  g(c1, n1, c2, f2) := g(c1, n1, c2, f2) ⊔ s
  // introduce unknown field values
  s' := ⊥State
  if u ≠ none then
    for all ℓ ∈ L, p ∈ P do
      if u(ℓ)(p) ≠ unknown then
        // the field has been referenced
        s'(ℓ)(p) := s(ℓ)(p)
      end if
    end for
  end if
  // propagate the resulting state into the function entry
  a.propagate'(c2, entry(f2), s')
  // propagate flow for the return edge, if any is known already
  let t = m(c2, exit(f2))
  if t ≠ none then
    a.funexit'(c1, n1, c2, f2, t)
  end if
```

Fig. 4. Algorithm for $\mathit{funentry}'(c_1, n_1, c_2, f_2, s)$. This modified version of *funentry* "lazily" propagates s into the abstract state at $\mathit{entry}(f_2)$ in context c_2. The abstract state s' is unknown for all fields that have not yet been referenced by the function being called according to u (recall that $\bot_{\mathsf{State}}$ maps all fields to unknown).

```
a.funexit'(c1 ∈ C, n1 ∈ N, c2 ∈ C, f2 ∈ F, s ∈ State):
  let (_, g) = a and u_g = g(c1, n1, c2, f2)
  s' := ⊥State
  for all ℓ ∈ L, p ∈ P do
    if s(ℓ)(p) = unknown then
      // the field has not been accessed, so restore its value from the call edge state
      s'(ℓ)(p) := u_g(ℓ)(p)
    else
      s'(ℓ)(p) := s(ℓ)(p)
    end if
  end for
  a.propagate'(c1, after(n1), s')
```

Fig. 5. Algorithm for $\mathit{funexit}'(c_1, n_1, c_2, f_2, s)$. This modified version of *funexit* restores field values that have not been accessed within the function being called, using the value from before the call. It then propagates the resulting state as in the original operation.

Figure 6 illustrates the dataflow at function entries and exits as modeled by the $\mathit{funexit}'$ and $\mathit{funentry}'$ operations. The two nodes n_1 and n_2 represent function call statements that invoke the function f. Assume that the value of the field p in the abstract object ℓ, denoted $\ell.p$, is v_1 at n_1 and v_2 at n_2 where $v_1, v_2 \in \mathsf{Value}$. When dataflow first arrives at $\mathit{entry}(f)$ the $\mathit{funentry}'$ operation sets $\ell.p$ to unknown. Assuming that f does not access $\ell.p$ it remains unknown

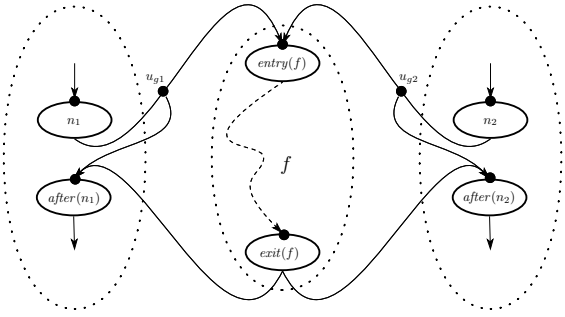

Fig. 6. A function f being called from two different statements, n_1 and n_2 appearing in other functions (for simplicity, all with the same context c). The edges indicate dataflow, and each bullet corresponds to an element of State with $u_{g1} = g(c, n_1, c, f)$ and $u_{g2} = g(c, n_2, c, f)$ where $g \in \mathit{CallGraph}$.

throughout f, so $\mathit{funexit}'$ can safely restore the original value v_1 by merging the state from $\mathit{exit}(f)$ with u_{g1} (the state recorded at the call edge) at $\mathit{after}(n_1)$. Similarly for the other call site, the value v_2 will be restored at $\mathit{after}(n_2)$. Thus, the dataflow for non-referenced fields respects the interprocedurally valid paths. This is in contrast to the basic framework where the value of $\ell.p$ would be $v_1 \sqcup v_2$ at both $\mathit{after}(n_1)$ and $\mathit{after}(n_2)$. Thereby, the modification of *funexit* may – perhaps surprisingly – cause the resulting analysis solution to be more precise than in the basic framework even for non-unknown field values. If a statement in f writes a value v' to $\ell.p$ it will no longer be unknown, so v' will propagate to both $\mathit{after}(n_1)$ and $\mathit{after}(n_2)$. If the transfer function of a statement in f invokes $\mathit{getfield}'$ to obtain the value of $\ell.p$ while it is unknown, it will be recovered by considering the call edges into f, as explained in Section 3.3.

The *getstate* operation is not modified. A transfer function cannot notice the fact that the returned State elements may contain unknown field values, because it is not permitted to read a field value through such a state.

Finally, the *getcallgraph* operation requires a minor modification to ensure that its output has the same type although the underlying lattice has changed:

$a.\mathit{getcallgraph}'()$:
return $\{(c_1, n_1, c_2, f_2) \mid g(c_1, n_1, c_2, f_2) \neq \mathsf{none}\}$ **where** $(\_, g) = a$

To demonstrate how the lazy propagation framework manages to avoid certain redundant computations, consider again the `markAsRunnable` function in Section 2.5. Suppose that the analysis first encounters a call to this function with some abstract state s. This call triggers the analysis of the function body, which accesses only a few object fields within s. The abstract state at the entry location of the function is unknown for all other fields. If new flow subsequently arrives via a call to the function with another abstract state s' where $s \sqsubseteq s'$, the introduction of unknown values ensures that the function body is only reanalyzed if s' differs from s at the few relevant fields that are not unknown.

3.3 Recovering Unknown Field Values

We now turn to the definition of the auxiliary operation *recover*. It gets invoked by *getfield*′ and *propagate*′ whenever an unknown element needs to be replaced by a proper field value. The operation returns the desired field value but also, as a side effect, modifies the relevant abstract states for function entry locations in a.

The key observation for defining $recover(c, n, \ell, p)$ where $c \in C$, $n \in N$, $\ell \in L$, and $p \in P$ is that unknown is only introduced in *funentry*′ and that each call edge – very conveniently – records the abstract state just before the ordinary field value is changed into unknown. Thus, the operation needs to go back through the call graph and recover the missing information. However, it only needs to modify the abstract states that belong to function entry statements.

Recovery is a two phase process. The first phase constructs a directed multi-rooted graph G the nodes of which are a subset of $C \times F$. It is constructed from the call graph in a backward manner starting from (c, n) as the smallest graph satisfying the following two constraints, where $(m, g) = a$:

- If $u(\ell)(p) = \mathsf{unknown}$ where $u = m(c, entry(fun(n)))$ then G contains the node $(c, fun(n))$.
- For each node (c_2, f_2) in G and for each (c_1, n_1) where $g(c_1, n_1, c_2, f_2) \neq \mathsf{none}$:
  - If $u_g(\ell)(p) = \mathsf{unknown} \wedge u_1(\ell)(p) = \mathsf{unknown}$ where $u_g = g(c_1, n_1, c_2, f_2)$ and $u_1 = m(c_1, entry(fun(n_1)))$ then G contains the node $(c_1, fun(n_1))$ with an edge to (c_2, f_2),
  - otherwise, (c_2, f_2) is a root of G.

The resulting graph is essentially a subgraph of the call graph such that every node (c', f') in G satisfies $u(\ell)(p) = \mathsf{unknown}$ where $u = m(c', entry(f'))$. A node is a root if at least one of its incoming edges contributes with a non-unknown value. Notice that root nodes may have incoming edges.

The second phase is a fixpoint computation over G:

```
// recover the abstract value at the roots of G
for each root (c', f') of G do
  let u' = m(c', entry(f'))
  for all (c1, n1) where g(c1, n1, c', f') ≠ none do
    let u_g = g(c1, n1, c', f') and u1 = m(c1, entry(fun(n1)))
    if u_g(ℓ)(p) ≠ unknown then
      u'(ℓ)(p) := u'(ℓ)(p) ⊔ u_g(ℓ)(p)
    else if u1(ℓ)(p) ≠ unknown then
      u'(ℓ)(p) := u'(ℓ)(p) ⊔ u1(ℓ)(p)
    end if
  end for
end for
// propagate throughout G at function entry nodes
S := the set of roots of G
while S ≠ ∅ do
  select and remove (c', f') from S
```

```
    let u' = m(c', entry(f'))
    for each successor (c2, f2) of (c', f') in G do
      let u2 = m(c2, entry(f2))
      if u'(ℓ)(p) ⋢ u2(ℓ)(p) then
        u2(ℓ)(p) := u2(ℓ)(p) ⊔ u'(ℓ)(p)
        add (c2, f2) to S
      end if
    end for
  end while
```

This phase recovers the abstract value at the roots of G and then propagates the value through the nodes of G until a fixpoint is reached. Although *recover* modifies abstract states in this phase, it does not modify the worklist. After this phase, we have $u(\ell)(p) \neq \mathsf{unknown}$ where $u = m(c', \mathit{entry}(f'))$ for each node (c', f') in G. (Notice that the side effects on a only concern abstract states at function entry statements.) In particular, this holds for $(c, \mathit{fun}(n))$, so when $\mathit{recover}(c, n, \ell, p)$ has completed the two phases, it returns the desired value $u(\ell)(p)$ where $u = m(c, \mathit{entry}(\mathit{fun}(n)))$.

Notice that the graph G is empty if $u(\ell)(p) \neq \mathsf{unknown}$ where $u = m(c, \mathit{entry}(\mathit{fun}(n)))$ (see the first of the two constraints defining G). In this case, the desired field has already been recovered, the second phase is effectively skipped, and $u(\ell)(p)$ is returned immediately.

Figure 7 illustrates an example of interprocedural dataflow among four functions. (This example ignores dataflow for function returns and assumes a fixed calling context c.) The statements write_1 and write_2 write to a field $\ell.p$, and *read* reads from it. Assume that the analysis discovers all the call edges before visiting *read*. In that case, $\ell.p$ will have the value unknown when entering f_2 and f_3, which will propagate to f_4. The transfer function for *read* will then invoke $\mathit{getfield}'$, which in turn invokes *recover*. The graph G will be constructed with three nodes: (c, f_2), (c, f_3), and (c, f_4) where (c, f_2) and (c, f_3) are roots and have edges to (c, f_4). The second phase of *recover* will replace the unknown value of $\ell.p$ at $\mathit{entry}(f_2)$ and $\mathit{entry}(f_2)$ by its proper value stored at the call edges and then propagate that value to $\mathit{entry}(f_3)$ and finally return it to $\mathit{getfield}'$. Notice that the value of $\ell.p$ at, for example, the call edges, remains unknown. However, if dataflow subsequently arrives via transfer functions of other statements, those unknown values may be replaced by ordinary values. Finally, note that this simple example does not require fixpoint iteration within *recover*, however that becomes necessary when call graphs contain cycles (resulting from programs with recursive function calls).

The modifications only concern the AnalysisLattice ADT, in terms of which all transfer functions of an analysis are defined. The transfer functions themselves are not changed. Although invocations of *recover* involve traversals of parts of the call graph, the main worklist algorithm (Figure 1) requires no modifications.

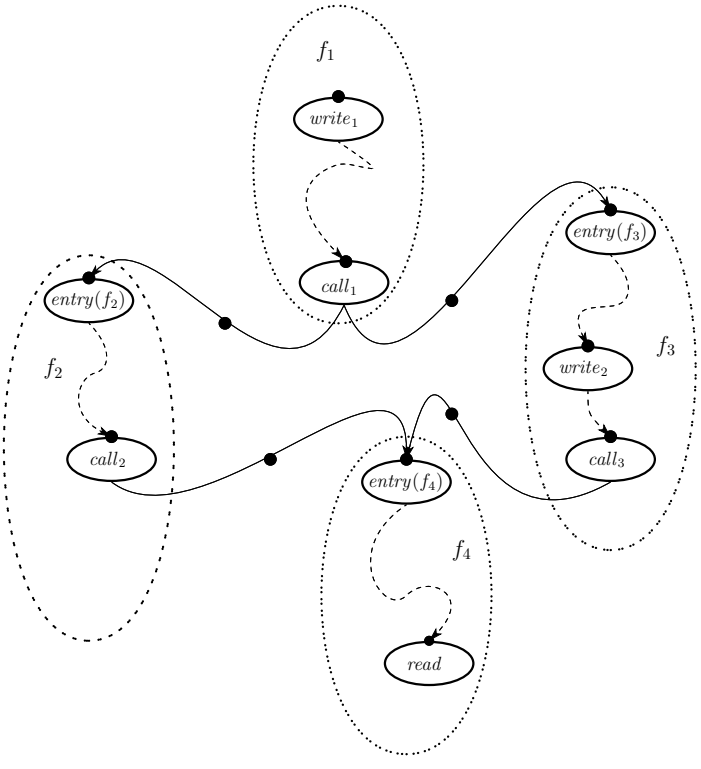

Fig. 7. Fragments of four functions, $f_1 \ldots f_4$. As in Figure 6, edges indicate dataflow and bullets correspond to elements of State. The statements $write_1$ and $write_2$ write to a field $\ell.p$, and *read* reads from it. The *recover* operation applied to the *read* statement and $\ell.p$ will ensure that values written at $write_1$ and $write_2$ will be read at the *read* statements, despite the possible presence of unknown values.

4 Implementation and Experiments

To examine the impact of lazy propagation on analysis performance, we extended the Java implementation of TAJS, our type analyzer for JavaScript [14], by systematically applying the modifications described in Section 3. As usual in dataflow analysis, primitive statements are grouped into basic blocks. The implementation focuses on the JavaScript language itself and the built-in library, but presently excludes the DOM API, so we use the most complex benchmarks from the V8[1] and SunSpider[2] benchmark collections for the experiments.

Descriptions of other aspects of TAJS not directly related to lazy propagation may be found in the TAJS paper [14]. These include the use of recency

[1] `http://v8.googlecode.com/svn/data/benchmarks/v1/`
[2] `http://www2.webkit.org/perf/sunspider-0.9/sunspider.html`

Table 1. Performance benchmark results

			Iterations			Time (seconds)			Memory (MB)		
	LOC	Blocks	*basic*	*basic+*	*lazy*	*basic*	*basic+*	*lazy*	*basic*	*basic+*	*lazy*
`richards.js`	529	478	2663	2782	1399	5.6	4.6	3.8	11.05	6.4	3.7
`benchpress.js`	463	710	18060	12581	5097	33.2	13.4	5.4	42.02	24.0	7.8
`delta-blue.js`	853	1054	∞	∞	63611	∞	∞	136.7	∞	∞	140.5
`cryptobench.js`	1736	2857	∞	43848	17213	∞	99.4	22.1	∞	127.9	42.8
`3d-cube.js`	342	545	7116	4147	2009	14.1	5.3	4.0	18.4	10.6	6.2
`3d-raytrace.js`	446	575	∞	30323	6749	∞	24.8	8.2	∞	16.7	10.1
`crypto-md5.js`	296	392	5358	1004	646	4.5	2.0	1.8	6.1	3.6	2.7
`access-nbody.js`	179	149	551	523	317	1.8	1.3	1.0	3.2	1.7	0.9

abstraction [4], which complicates the implementation, but does not change the properties of the lazy propagation technique.

We compare three versions of the analysis: *basic* corresponds to the basic framework described in Section 2; *basic+* extends the basic version with the copy-on-write and maybe-modified techniques discussed in Section 2.5, which is the version used in [14]; and *lazy* is the new implementation using lazy propagation (without the other extensions from the *basic+* version).

Table 1 shows for each program, the number of lines of code, the number of basic blocks, the number of fixpoint iterations for the worklist algorithm (Figure 1), analysis time (in seconds, running on a 3.2GHz PC), and memory consumption. We use ∞ to denote runs that require more than 512MB of memory.

We focus on the time and space requirements for these experiments. Regarding precision, *lazy* is in principle more precise than *basic+*, which is more precise than *basic*. On these benchmark programs, however, the precision improvement is insignificant with respect to the number of potential type related bugs, which is the precision measure we have used in our previous work.

The experiments demonstrate that although the copy-on-write and maybe-modified techniques have a significant positive effect on the resource requirements, lazy propagation leads to even better results. The results for `richards.js` are a bit unusual as it takes more iterations in *basic+* than in *basic*, however the fixpoint is more precise in *basic+*.

The benchmark results demonstrate that lazy propagation results in a significant reduction of analysis time without sacrificing precision. Memory consumption is reduced by propagating less information during the fixpoint computation and fixpoints are reached in fewer iterations by eliminating a cause of redundant computation observed in the basic framework.

5 Related Work

Recently, JavaScript and other scripting languages have come into the focus of research on static program analysis, partly because of their challenging dynamic nature. These works range from analysis for security vulnerabilities [29, 8] to static type inference [7, 27, 1, 14]. We concentrate on the latter category, aiming to develop program analyses that can compensate for the lack of static type

checking in these languages. The interplay of language features of JavaScript, including first-class functions, objects with modifiable prototype chains, and implicit type coercions, makes analysis a demanding task.

The IFDS framework by Reps, Horwitz, and Sagiv [23] is a powerful and widely used approach for obtaining precise interprocedural analyses. It requires the underlying lattice to be a powerset and the transfer functions to be distributive. Unfortunately, these requirements are not met by our type analysis problem for dynamic object-oriented scripting languages. The more general IDE framework also requires distributive transfer functions [25]. A connection to our approach is that fields that are marked as unknown at function exits, and hence have not been referenced within the function, are recovered from the call site in the same way local variables are treated in IFDS.

Sharir and Pnueli's functional approach to interprocedural analysis can be phrased both with symbolic representations and in an iterative style [26], where the latter is closer to our approach. With the complex lattices and transfer functions that appear to be necessary in analyses for object-oriented scripting languages, symbolic representations are difficult to work with, so TAJS uses the iterative style and a relatively direct representation of lattice elements. Furthermore, the functional approach is expensive if the analysis lattice is large.

Our analysis framework encompasses a general notion of context sensitivity through the C component of the analysis instances. Different instantiations of C lead to different kinds of context sensitivity, including variations of the call-string approach [26], which may also affect the quality of interprocedural analysis. We leave the choice of C open here; TAJS currently uses a heuristic that distinguishes call sites that have different values of `this`.

The introduction of unknown field values subsumes the *maybe-modified* technique that we used in the first version of TAJS [14]: a field whose value is unknown is definitely not modified. Both ideas can be viewed as instances of side effect analysis. Unlike, for example, the side effect analysis by Landi et al. [24] our analysis computes the call graph on-the-fly and we exploit the information that certain fields are found to be non-referenced for obtaining the lazy propagation mechanism. Via this connection to side effect analysis, one may also view the unknown field values as establishing a frame condition as in separation logic [21].

Combining call graph construction with other analyses is common in pointer alias analysis with function pointers, for example in the work of Burke et al. [11]. That paper also describes an approach called deferred evaluation for increasing analysis efficiency, which is specialized to flow insensitive alias analysis.

Lazy propagation is related to lazy evaluation (e.g., [22]) as it produces values passed to functions on demand, but there are some differences. Lazy propagation does not defer evaluation as such, but just the propagation of the values; it applies not just to the parameters but to the entire state; and it restricts laziness to data structures (values of fields).

Lazy propagation is different from demand-driven analysis [13]. Both approaches defer computation, but demand-driven analysis only computes results for selected hot spots, whereas our goal is a whole-program analysis that infers

information for all program points. Other techniques for reducing the amount of redundant computation in fixpoint solvers is difference propagation [6] and use of interprocedural def-use chains [28]. It might be possible to combine those techniques with lazy propagation, although they are difficult to apply to the complex transfer functions that we have in type analysis for JavaScript.

6 Conclusion

We have presented *lazy propagation* as a technique for improving the performance of interprocedural analysis in situations where existing methods, such as IFDS or the functional approach, do not apply. The technique is described by a systematic modification of a basic iterative framework. Through an implementation that performs type analysis for JavaScript we have demonstrated that it can significantly reduce the memory usage and the number of fixpoint iterations without sacrificing analysis precision. The result is a step toward sound, precise, and fast static analysis for object-oriented languages in general and scripting languages in particular.

Acknowledgments. The authors thank Stephen Fink, Michael Hind, and Thomas Reps for their inspiring comments on early versions of this paper.

References

1. Anderson, C., Giannini, P., Drossopoulou, S.: Towards type inference for JavaScript. In: Black, A.P. (ed.) ECOOP 2005. LNCS, vol. 3586, pp. 428–452. Springer, Heidelberg (2005)
2. Artzi, S., Kiezun, A., Dolby, J., Tip, F., Dig, D., Paradkar, A.M., Ernst, M.D.: Finding bugs in dynamic web applications. In: Proc. International Symposium on Software Testing and Analysis, ISSTA 2008. ACM, New York (July2008)
3. Atkinson, D.C., Griswold, W.G.: Implementation techniques for efficient data-flow analysis of large programs. In: Proc. International Conference on Software Maintenance, ICSM 2001, pp. 52–61 (November 2001)
4. Balakrishnan, G., Reps, T.W.: Recency-abstraction for heap-allocated storage. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 221–239. Springer, Heidelberg (2006)
5. Chase, D.R., Wegman, M., Kenneth Zadeck, F.: Analysis of pointers and structures. In: Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 1990 (June 1990)
6. Fecht, C., Seidl, H.: Propagating differences: An efficient new fixpoint algorithm for distributive constraint systems. In: Hankin, C. (ed.) ESOP 1998. LNCS, vol. 1381, p. 90. Springer, Heidelberg (1998)
7. Furr, M., An, Jong hoon (David), Foster, J.S., Hicks, M.W.: Static type inference for Ruby. In: Jacobson Jr., M.J., Rijmen, V., Safavi-Naini, R. (eds.) SAC 2009. LNCS, vol. 5867, Springer, Heidelberg (2009)

8. Guha, A., Krishnamurthi, S., Jim, T.: Using static analysis for Ajax intrusion detection. In: Proc. 18th International Conference on World Wide Web, WWW 2009 (2009)
9. Heidegger, P., Thiemann, P.: Recency types for analyzing scripting languages. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 200–224. Springer, Heidelberg (2010)
10. Hind, M.: Pointer analysis: haven't we solved this problem yet? In: Proc. ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE 2001, pp. 54–61 (June 2001)
11. Hind, M., Burke, M.G., Carini, P.R., Choi, J.-D.: Interprocedural pointer alias analysis. ACM Transactions on Programming Languages and Systems 21(4), 848–894 (1999)
12. Horwitz, S., Demers, A., Teitebaum, T.: An efficient general iterative algorithm for dataflow analysis. Acta Informatica 24(6), 679–694 (1987)
13. Horwitz, S., Reps, T., Sagiv, M.: Demand interprocedural dataflow analysis. In: Proc. 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering, FSE 1995 (October 1995)
14. Jensen, S.H., Møller, A., Thiemann, P.: Type analysis for JavaScript. In: Palsberg, J., Su, Z. (eds.) SAS 2009. LNCS, vol. 5673, pp. 238–255. Springer, Heidelberg (2009)
15. Jensen, S.H., Møller, A., Thiemann, P.: Interprocedural analysis with lazy propagation. Technical report, Department of Computer Science, Aarhus University (2010), `http://cs.au.dk/~amoeller/papers/lazy/`
16. Jones, N.D., Muchnick, S.S.: A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In: Proc. 9th ACM Symposium on Principles of Programming Languages, POPL 1982 (January 1982)
17. Kam, J.B., Ullman, J.D.: Global data flow analysis and iterative algorithms. Journal of the ACM 23(1), 158–171 (1976)
18. Kam, J.B., Ullman, J.D.: Monotone data flow analysis frameworks. Acta Informatica 7, 305–317 (1977)
19. Kildall, G.A.: A unified approach to global program optimization. In: Proc. 1st ACM Symposium on Principles of Programming Languages. In: POPL 1973 (October 1973)
20. Liskov, B., Zilles, S.N.: Programming with abstract data types. ACM SIGPLAN Notices 9(4), 50–59 (1974)
21. O'Hearn, P.W., Reynolds, J.C., Yang, H.: Local reasoning about programs that alter data structures. In: Fribourg, L. (ed.) CSL 2001 and EACSL 2001. LNCS, vol. 2142, p. 1. Springer, Heidelberg (2001)
22. Jones, S.L.P.: The Implementation of Functional Programming Languages. Prentice Hall, Englewood Cliffs (1987)
23. Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In: Proc. 22th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1995, pp. 49–61 (January 1995)
24. Ryder, B.G., Landi, W., Stocks, P., Zhang, S., Altucher, R.: A schema for interprocedural modification side-effect analysis with pointer aliasing. ACM Transactions on Programming Languages and Systems 23(2), 105–186 (2001)

25. Sagiv, S., Reps, T.W., Horwitz, S.: Precise interprocedural dataflow analysis with applications to constant propagation. Theoretical Computer Science 167(1&2), 131–170 (1996)
26. Sharir, M., Pnueli, A.: Two approaches to interprocedural dataflow analysis. In: Program Flow Analysis: Theory and Applications, pp. 189–233. Prentice-Hall, Englewood Cliffs (1981)
27. Thiemann, P.: Towards a type system for analyzing JavaScript programs. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 408–422. Springer, Heidelberg (2005)
28. Tok, T.B., Guyer, S.Z., Lin, C.: Efficient flow-sensitive interprocedural data-flow analysis in the presence of pointers. In: Mycroft, A., Zeller, A. (eds.) CC 2006. LNCS, vol. 3923, pp. 17–31. Springer, Heidelberg (2006)
29. Xie, Y., Aiken, A.: Static detection of security vulnerabilities in scripting languages. In: Proc. 15th USENIX Security Symposium (August 2006)