Sparse Dataflow Analysis with Pointers and Reachability

Magnus Madsen and Anders Møller

Aarhus University, Denmark
{magnusm,amoeller}@cs.au.dk

Abstract. Many static analyzers exploit sparseness techniques to reduce the amount of information being propagated and stored during analysis. Although several variations are described in the literature, no existing technique is suitable for analyzing JavaScript code. In this paper, we point out the need for a sparse analysis framework that supports pointers and reachability. We present such a framework, which uses static single assignment form for heap addresses and computes def-use information on-the-fly. We also show that essential information about dominating definitions can be maintained efficiently using quadtrees. The framework is presented as a systematic modification of a traditional dataflow analysis algorithm.
Our experimental results demonstrate the effectiveness of the technique for a suite of JavaScript programs. By also comparing the performance with an idealized staged approach that computes pointer information with a pre-analysis, we show that the cost of computing def-use information on-the-fly is remarkably small.

1 Introduction

Previous work on dataflow analysis has demonstrated that sparse analysis is a powerful technique for improving performance of many kinds of static analysis without sacrificing precision [7,8,14,15,20,21], compared to more basic dataflow analysis frameworks [12, 13]. The key idea in sparse analysis is that dataflow should be propagated directly from definitions to uses in the program code, unlike "dense" analysis that propagates dataflow along the control-flow. A potential advantage of sparse analysis is that it propagates and stores only relevant information, not entire abstract states. Another advantage is that transfer functions need only be recomputed when their dependencies change.

While developing analysis tools for JavaScript we have found that the existing approaches described in the literature for building sparse analyses do not apply to the language features and common programming patterns that appear in JavaScript code. Specifically, context-sensitive branch pruning (a variant of unreachable code elimination by Wegman and Zadeck [21]) is an important analysis technique, as explained below, for handling the use of function overloading, which in JavaScript is programmed using reflection. Moreover, the common wisdom from analysis of e.g. Java code that context-insensitive analysis is usually faster than context-sensitive analysis [18] apparently does not apply to JavaScript code, which, as discussed below, makes it difficult to design practically useful staged sparse analyses for this language.

As a motivating example, consider the JavaScript function on the right that exhibits a simple form of overloading. Here, the branch condition `b.p` decides whether the `f` function should have one behavior or another (in real-world JavaScript code, complex function overloading is mimicked using various kinds of reflection in branch conditions, but the pattern is the same). It is often the case that the branch condition is *determinate* relative to the call context [17]. That is, in one call context, `b.p` is known to be true, and in another call context, it is known to be false. In a context-sensitive dense analysis, this is no problem for the precision: `f` is simply analyzed in two contexts, corresponding to the two cases, such that the analysis logically clones `f` and analyzes it twice. When dataflow reaches the `if` statement, the analysis can then discover that one branch is dead and only propagate dataflow along the live branch.

```
function f(b, x, y) {
  var r;
  if (b.p) {
    r = x.a;
  } else {
    r = y.a;
  }
  return r;
}
```

To reason precisely about such program code, for example, with the purpose of computing call graphs or information about types of expressions, a static analysis must account for *reachability*, i.e. whether branches are live or dead in the individual contexts. At the same time it must handle heap allocated storage, as objects are pervasive in JavaScript. Moreover, even apparently simple operations in JavaScript, such as reading an object property, are complex procedures that involve e.g. type coercion and traversal of dynamically constructed prototype chains. This makes it beneficial to design analysis techniques that support complex transfer functions, for example, all computable monotone functions [12]. It is well known how to accomplish all this using dense analysis (see e.g. the TAJS analysis [9]). Our goal is to take the step to sparse analysis, without sacrificing precision compared to the original dense version.

One way to build sparse analyses for programs with pointers is to use a staged approach where a pre-analysis computes a sound approximation of the memory addresses that are defined or used at each operation in the code, and then establish def-use edges that the main analysis can use for sparse flow-sensitive dataflow propagation [8, 14]. Unfortunately, this does not work in our setting, unless the pre-analysis is as precise as the main analysis (and in that case, there would be no need for the main analysis, obviously): If the pre-analysis is flow-insensitive, for example, it would establish def-use edges from both `x.a` and `y.a` to `r` in our example, which would destroy the precision of the main analysis. Note that one of the main results of Oh et al. [14] is that, in their setting, approximations in the pre-analysis may lead to less sparseness but it will never affect the precision of the main analysis (due to their use of data dependence instead of def-use chains). However, for a language like JavaScript where most operations may throw exceptions, their algorithm largely degrades to a dense analysis if reachability is involved. In another line of work, Tok et al. [20] and Chase et al. [2] compute def-use edges on-the-fly rather than using a pre-analysis, but also without taking reachability into account. Conversely, the sparse conditional constant analysis by Wegman and Zadeck [21] handles reachability, but not pointers. In summary, no existing technique satisfies the needs for making sparse analysis for languages like JavaScript.

Our contributions are as follows:

– We present the first algorithm for sparse dataflow analyses that supports pointers, reachability, and arbitrary monotone transfer functions, while preserving the precision of the corresponding dense analysis, and without requiring a pre-analysis to compute def-use information.
– We describe experimental results, based on a dataflow analysis for JavaScript, that show a considerable performance improvement when using sparse analysis compared to a traditional dense approach, which demonstrates that it is possible to perform efficient sparse analysis in a setting that involves pointers and reachability.
– We show experimentally that the overhead of computing dominating definitions on-the-fly is small, which makes our approach preferable to staged approaches that compute that information with a pre-analysis.
– We demonstrate that quadtrees are a suitable data structure for maintaining essential information about dominating definitions in sparse analysis.

We explain the technique as a framework where we can switch from dense to sparse analysis, without affecting the abstract domains or the transfer functions.

2 A Basic Analysis Framework

Our starting point is a variant of the classical monotone framework for flow-sensitive dataflow analysis [12] where programs are represented as control-flow graphs with abstract states associated with the entry and exit program points of each node. For simplicity this presentation focuses on intraprocedural analysis, although our implementation supports interprocedural analysis as discussed in Section 4.

We assume that we are given a control-flow graph where each node represents a statement $s \in S$, together with a set of abstract memory addresses $a \in A$, a lattice of abstract values $v \in V$, and a transfer function T_s for each statement s. The transfer functions are assumed to be expressed using the following three primitive operations:

– $\textsc{Read}(s \in S, a \in A) : V$. Returns the value v at the address a at the program point immediately before the statement s.
– $\textsc{Write}(v \in V, s \in S, a \in A)$. Writes the value v to the address a at the program point immediately after the statement s. Note that an invocation $\textsc{Write}(v, s, a)$ models a strong update [2]; if a weak update is desired, the transfer function should invoke $\textsc{Write}(v \sqcup \textsc{Read}(s, a), s, a)$.

– Continue($s_{src} \in S, s_{dst} \in S$). Indicates that the transfer function $T_{s_{src}}$ has completed and that s_{dst} is a possible successor, in other words that s_{dst} is reachable from s_{src}. (For example, this allows the transfer function for an `if` statement to selectively propagate dataflow to one of its branches.)

As conventional, the framework applies the transfer functions using an iterative worklist algorithm, starting from a designated program entry statement, until the global fixpoint is reached. The ordering of the worklist W is left unspecified, so the analysis implementor may freely choose any. We assume the lattice V has finite height; for simplicity we ignore widening.

In the case of JavaScript, most transfer functions are complex operations that involve multiple Read and Write operations. For example, the transfer function for a simple assignment `x = y.p` in general requires traversal of scope chains and prototype chains. This can be accomplished as shown in previous work on the TAJS analysis [9]. Although that analysis uses more elaborate abstract domains, it can in principle all be expressed within the present framework.

A traditional dense propagation strategy [12] maintains an entire abstract state at each program point as a map from addresses to values:

$I : S \times A \to V$ is the map of *incoming states*
$O : S \times A \to V$ is the map of *outgoing states*

Reading from an address is then simply a matter of looking up its value in the abstract state in I, and writing similarly updates O. (In practice, analysis implementations often maintain only O, since the information in I can be inferred when needed; we include both maps explicitly to simplify the presentation in Section 3.) Continuing from s_{src} to s_{dst} is handled by joining the entire outgoing state at s_{src} into the incoming state at s_{dst}. To initiate the analysis, I and O return the bottom element $\perp$ of V for every statement and address, except that we assume an entry statement s_{entry} with a no-op transfer function (that just calls Continue) and where $I(s_{entry}, a)$ and $O(s_{entry}, a)$ both describe the initial abstract state for every address a. The initial worklist is then $W = \{s_{entry}\}$.

More formally, reading the value of an address $a \in A$ at statement $s \in S$ is implemented simply by looking up the value in the incoming state:

```
Read($s \in S, a \in A$) : $V$
1  return $I(s, a)$
```

Similarly, writing a value $v \in V$ to the address $a \in A$ at statement $s \in S$ is implemented by writing to the outgoing state:

```
Write($v \in V, s \in S, a \in A$)
1  $O(s, a) := v$
```

Propagation of dataflow from statement $s_{src} \in S$ to $s_{dst} \in S$ is implemented by joining all the values from the outgoing state of s_{src} into the incoming state of s_{dst}. If a value is changed then s_{dst} is added to the worklist. Reachability is implicitly supported since an unreachable statement has every value set to the bottom element $\bot$, whereas we assume that every reachable statement will have at least one value set to non-bottom, and so propagation from a reachable statement to an unreachable statement will always cause the unreachable statement to be added to the worklist:

```
CONTINUE($s_{src} \in S, s_{dst} \in S$)
1  for each $a \in A$
2      let $v = O(s_{src}, a)$
3      let $v' = I(s_{dst}, a)$
4      if $v \not\sqsubseteq v'$
5          $I(s_{dst}, a) := v \sqcup v'$
6          $W := W \cup \{s_{dst}\}$
```

The main fixpoint computation is implemented by the SOLVE procedure. It maintains a global worklist W of pending statements and iteratively extracts a statement and evaluates its transfer function, which may cause new statements to be added to the worklist. The fixpoint is found when the worklist is empty:

```
SOLVE($E : A \to V$), where $E$ is the entry state
1  $I(s, a) := O(s, a) := \bot$ for all $s \in S$, $a \in A$
2  $I(s_{entry}, a) := O(s_{entry}, a) := E(a)$ for all $a \in A$
3  $W := \{s_{entry}\}$
4  while $W \neq \emptyset$
5      let $s$ = DEQUEUE($W$)
6      $O(s, a) := I(s, a)$ for all $a \in A$
7      apply the transfer function $T_s$
```

3 Sparse Analysis

We now show how the basic analysis framework from the preceding section can be changed into our sparse analysis technique. As a first step, we modify the definitions of the incoming and outgoing states to become partial maps, $I : S \times A \hookrightarrow V$ and $O : S \times A \hookrightarrow V$, since we now want to maintain values only for the statements and addresses that are involved in READ or WRITE operations, respectively. Next, we add four new components that are all built incrementally during the fixpoint computation:

$R \subseteq S \times S$ is the set of *reachable edges*
$P : S \times A \hookrightarrow V$ specifies the placement and values of *ϕ-nodes*
$DU \subseteq S \times A \times S$ is the set of *def-use edges*
$F : S \times S \to \mathcal{P}(A)$ is the map of *frontier addresses*

The R component now explicitly tracks the set of reachable edges in the control-flow graph: if CONTINUE(s_{src}, s_{dst}) has been invoked, then $(s_{src}, s_{dst}) \in R$. As in previous sparse analysis techniques, we use SSA (static single assignment form) to ensure that each use site has a unique associated definition site [4,7,21]. When $P(s, a)$ is defined with some value v, the statement s plays the role of a ϕ-node for address a, where v is then the merged value from the incoming dataflow. As effect we obtain SSA for all addresses, not only for local variables. Each triple $(s_1, a, s_2) \in DU$ represents a def-use edge, where s_1 is a definition site or a ϕ-node and s_2 is a use site or a ϕ-node for a.

Since the analysis discovers definition sites and use sites incrementally, the set of def-use edges changes during the analysis. The F map supports this construction of def-use edges whenever frontier edge becomes reachable, as explained later in this section.

The Solve procedure is unmodified, except that line 6 is omitted in the sparse analysis version. The remainder of this section explains the modifications of the Read, Write, and Continue procedures.

Notation and terminology We view maps as mutable dictionaries. For example, if $f : A \to B$ is a map, then $f(x) := v$ denotes the update of f such that subsequently $f(x) = v$. If $f : X \hookrightarrow Y$ is a partial map, then $f_\star$ denotes the domain of f, i.e. the subset of X where f is defined. We assume the reader is familiar with the concepts of *SSA*, *dominance frontiers*, and *dominator trees* from e.g. Cytron et al. [4]. Specifically, a statement s_2 is in the *dominance frontier* of a statement s_0 if s_0 dominates some predecessor s_1 of s_2 in the control-flow graph, but s_0 does not dominate s_2. The edge (s_1, s_2) is then called a *frontier edge* of s_0. We say that a statement s is a *ϕ-node* (resp. *definition site* or *use site*) for an address a if $(s, a) \in P_\star$ (resp. $(s, a) \in O_\star$ or $(s, a) \in I_\star$). Only merge points in the control-flow graph can be used as ϕ-nodes. For simplicity, we assume that the statements at merge points are no-ops, such that they cannot be definition sites or use sites (thus, $P_\star$ and $I_\star \cup O_\star$ are disjoint).

The following key invariants are maintained by the Read, Write, and Continue operations in the sparse analysis framework:

[*flow*] If $(s_1, a, s_2) \in DU$ for some statements s_1, s_2 and some address a, then
- either $O(s_1, a)$ or $P(s_1, a)$ is defined with some value v,
- either $I(s_2, a)$ or $P(s_2, a)$ is defined with some value v', and
- the value of a at s_1 has been propagated to s_2, i.e. $v \sqsubseteq v'$.

[*def-use*] If (and only if) a statement s_1 is a definition site or ϕ-node for some address a, i.e. $(s_1, a) \in O_\star \cup P_\star$, s_k is a use site or ϕ-node for a, i.e. $(s_k, a) \in I_\star \cup P_\star$, such that s_1 dominates s_k and there is a path $s_1, s_2, \ldots, s_k$ where each step is reachable, i.e. $(s_i, s_{i+1}) \in R$ for all i, and moreover, there is no definition site or ϕ-node for a between s_1 and s_k, i.e. $(s_i, a) \notin I_\star \cup P_\star$ for all $i = 2, 3, \ldots, k-1$, then there exists a def-use edge $(s_1, a, s_k) \in DU$.

[*phi-use*] If s is a ϕ-node for a, i.e. $(s, a) \in P_\star$, then for every reachable incoming control-flow graph edge $(s_1, s) \in R$ there is a def-use edge $(s_2, a, s) \in DU$ where s_2 is the nearest dominator of s_1 and s_2 is a definition site or ϕ-node for a.

[*phi*] If a statement s_0 is a definition site or ϕ-node for some address a, i.e. $(s_0, a) \in O_\star \cup P_\star$, then for every frontier edge (s_1, s_2) of s_0 that is reachable, i.e. $(s_1, s_2) \in R$, the statement s_2 is a ϕ-node for a, i.e. $(s_2, a) \in P_\star$.

[*frontier*] If $a \in F(s_1, s_2)$ for some statements s_1, s_2 and some address a, then (s_1, s_2) is a frontier edge of a dominator s_0 of s_1 that defines a, i.e. $(s_0, a) \in O_\star \cup P_\star$.

Intuitively, the [*flow*] invariant ensures that dataflow has always been propagated along the existing def-use edges; [*def-use*] expresses the main requirements for construction of def-use edges, in particular that def-use edges respect reachability and dominance of definitions; [*phi-use*] ensures that def-use edges to ϕ-nodes also exist for all reachable incoming edges; [*phi*] ensures that ϕ-nodes are created along reachable dominance frontiers; and [*frontier*] expresses that F records which addresses are relevant for contructing def-use edges whenever a frontier edge becomes reachable.

3.1 Reading Values

The Read(s, a) operation retrieves the requested value from the incoming state if s is already known to be a use site for a. If a new use is discovered, the appropriate def-use edge must be introduced and the value propagated to s:

```
Read($s \in S, a \in A$) : $V$
1 if $(s, a) \notin I_\star$
2     $I(s, a) := \bot$
3     let $s_1$ = FindDef($s', a$) where $s'$ is the immediate dominator of $s$
4     $DU := DU \cup \{(s_1, a, s)\}$
5     Propagate($s_1, a, s$)
6 return $I(s, a)$
```

The FindDef procedure searches up the dominator tree to find the nearest definition site or ϕ-node for a:

```
FindDef($s \in S, a \in A$) : $S$
1 if $(s, a) \in O_\star \cup P_\star$
2     return $s$
3 else
4     return FindDef($s', a$) where $s'$ is the immediate dominator of $s$
```

We show in Section 3.4 how FindDef can be implemented more efficiently than this pseudo-code suggests. Also note that by initializing $I(s_{entry}, a)$ and $O(s_{entry}, a)$ for every address a according to the initial abstract state when the analysis starts, Read and FindDef are well-defined because s_{entry} is the root of the dominator tree.

The Propagate procedure, which is also used by the Write operation later, propagates a single value from a definition site or ϕ-node to a use site or ϕ-node, in order to satisfy the [*flow*] invariant. If the destination is a use site and its incoming state changes, then that statement is added to the worklist W. If the destination is a ϕ-node then propagation is invoked recursively for all its outgoing def-use edges:

```
PROPAGATE($s_{src} \in S, a \in A, s_{dst} \in S$)
if $(s_{src}, a) \in O_\star$
    let $v = O(s_{src}, a)$
else // must have $(s_{src}, a) \in P_\star$
    let $v = P(s_{src}, a)$
if $(s_{dst}, a) \in I_\star$
    let $v_{old} = I(s_{dst}, a)$
    if $v \not\sqsubseteq v_{old}$
        $I(s_{dst}, a) := v \sqcup v_{old}$
        $W := W \cup \{s_{dst}\}$
else // must have $(s_{dst}, a) \in P_\star$
    let $v_{old} = P(s_{dst}, a)$
    if $v \not\sqsubseteq v_{old}$
        $P(s_{dst}, a) := v \sqcup v_{old}$
        for each $s$ where $(s_{dst}, a, s) \in DU$
            PROPAGATE($s_{dst}, a, s$)
```

Notice that recursive calls to PROPAGATE can only happen along chains of def-use edges between ϕ-nodes, which are placed only at merge points, so the recursion is bounded by the block nesting depth of the program being analyzed.

3.2 Writing Values

The WRITE operation writes the given value to the outgoing state. If a new definition site is discovered, the set of def-use edges must be updated. Moreover, the written value is propagated along the outgoing def-use edges:

```
WRITE($v \in V, s \in S, a \in A$)
1  if $(s, a) \notin O_\star$
2      UPDATE($s, a$)
3      $O(s, a) := v$
4      FORWARD($s, a$)
5  else
6      $O(s, a) := v$
7  for each $s_{dst}$ where $(s, a, s_{dst}) \in DU$
8      PROPAGATE($s, a, s_{dst}$)
```

Whenever a new definition site is discovered in WRITE (line 1), def-use edges that bypass the new definition site and have the same address must be updated (line 2) and ϕ-nodes must be introduced at the iterated dominance frontiers along with associated def-use edges (line 4).

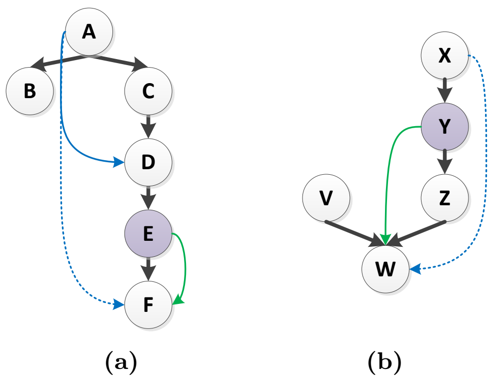

Fig. 1. Two control-flow graph fragments (with thick edges representing control-flow) that illustrate the UPDATE procedure. **(a)** The statement A defines some address that is used at both statements D and F (corresponding to the def-use edges A $\rightarrow$ D and A $\rightarrow$ F). At some point a definition is discovered at E. The dominating definition at E is A. Its use at D is *not* dominated by E and thus not affected by the new definition at E. The use at F, however, is dominated by E, so the def-use edge A $\rightarrow$ F is replaced by E $\rightarrow$ F. **(b)** The statement X defines some address that is used by the ϕ-node at W (corresponding to the def-use edge X $\rightarrow$ W). If a new definition is discovered at Y then X $\rightarrow$ W must be replaced by Y $\rightarrow$ W since X dominates Y.

```
UPDATE($s \in S, a \in A$)
1  let $s_1$ = FINDDEF($s, a$)
2  for each $s_2$ where $(s_1, a, s_2) \in DU$
3      if $s$ strictly dominates $s_2$
4          $DU := (DU \setminus \{(s_1, a, s_2)\}) \cup \{(s, a, s_2)\}$
5  for each $(s_3, s_4) \in S \times S$ that is a frontier edge of $s$
6      let $s_0$ = FINDDEF($s_3, a$)
7      if $s_0$ strictly dominates $s$
8          if $(s_0, a, s_4) \in DU$
9              $DU := (DU \setminus \{(s_0, a, s_4)\}) \cup \{(s, a, s_4)\}$
```

The UPDATE procedure updates the def-use edges that bypass the new definition. The first part (lines 1–4) handles the def-use edges that end at a statement dominated by the new definition, to restore the [*def-use*] invariant as illustrated in Figure 1(a); the second part (lines 5–9) handles the def-use edges that end at a dominance frontier node of the new definition, corresponding to [*phi-use*] as illustrated in Figure 1(b).

Note that DU does not always grow monotonically, since UPDATE both adds and removes edges. Termination is still ensured: a def-use edge (s_1, a, s_2) is only removed if a new definition site s_d is discovered such that s_d dominates s_2. Definitions are never removed, so the edge (s_1, a, s_2) can never be re-added, and only a finite number of def-use edges can be created. All other components in the sparse framework are monotonically increasing during the fixpoint computation.

The purpose of the Forward procedure is to introduce ϕ-nodes at the iterated dominance frontiers, together with def-use edges for the corresponding reachable frontier edges, and maintain the [*frontier*] invariant:

```
Forward($s \in S, a \in A$)
1 for each $(s_1, s_2) \in S \times S$ that is a frontier edge of $s$
2     if $a \notin F(s_1, s_2)$
3         $F(s_1, s_2) := F(s_1, s_2) \cup \{a\}$
4         if $(s_1, s_2) \in R$
5             MakePhi($s_2, a$)
```

Although a statement typically has a single frontier edge, it is possible to have multiple, for example, in connection to statements that may throw exceptions. Line 3 in Forward adds a to the frontier addresses of the frontier edge (s_1, s_2), which indicates that a has been defined by a statement that dominates s_1. If that edge is already known to be reachable, we may need to add a new ϕ-node at the frontier, which is handled by MakePhi as explained next.

```
MakePhi($s \in S, a \in A$)
1 if $(s, a) \notin P_\star$
2     Update($s, a$)
3     $P(s, a) := \bot$
4 for each $s_1 \in S$ where $s_1$ is an immediate predecessor of $s$ in the control-flow graph
5     if $(s_1, s) \in R$
6         let $s_2 =$ FindDef($s_1, a$)
7         $DU := DU \cup \{(s_2, a, s)\}$
8         Propagate($s_2, a, s$)
9 Forward($s, a$)
```

The MakePhi procedure is only invoked with s being a dominance frontier node. If s is not already a ϕ-node for a (line 1), we mark it as one (line 3). However, since a ϕ-node has a similar effect as a definition site, Update is called first to update the def-use edges, c.f. line 2 in Write. A ϕ-node also has a similar effect as a use site, although generally with multiple incoming def-use edges. For this reason, we make sure a def-use edge exists for every reachable income edge (lines 4–8), c.f. lines 3–5 in Read. Finally, the process is continued recursively for the iterated dominance frontiers (line 9).

3.3 Propagating Reachability

As the propagation of dataflow values is performed along def-use edges by Propagate, the primary role of Continue is to propagate reachability:

```
Continue($s_{src} \in S, s_{dst} \in S$)
1 if $(s_{src}, s_{dst}) \notin R$
2     $R := R \cup \{(s_{src}, s_{dst})\}$
3     $W := W \cup \{s_{dst}\}$
4     for each $a \in A$ where $a \in F(s_{src}, s_{dst}) \lor (s_{dst}, a) \in P_\star$
5         MakePhi($s_{dst}, a$)
```

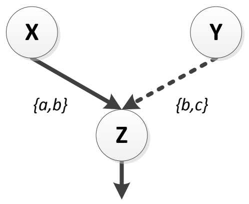

Fig. 2. A control-flow graph fragment, where the frontier edges $\mathsf{X} \rightarrow \mathsf{Z}$ and $\mathsf{Y} \rightarrow \mathsf{Z}$ hold the addresses $\{a, b\}$ and $\{b, c\}$, respectively. The edge $\mathsf{X} \rightarrow \mathsf{Z}$ is already reachable, and ϕ-nodes for $\{a, b\}$ at Z have already been introduced. Now CONTINUE is invoked for the edge $\mathsf{Y} \rightarrow \mathsf{Z}$. CONTINUE ensures that appropriate def-uses edge to Z are introduced for all the addresses a, b, and c, and for all the incoming edges to Z.

If the given control-flow graph edge (s_{src}, s_{dst}) is not already reachable, we mark it as reachable (line 2) and add s_{dst} to the worklist. However, this may trigger calls to MAKEPHI in two situations, corresponding to the two cases in line 4: The condition $a \in F(s_{src}, s_{dst})$ signals that (s_{src}, s_{dst}) is a frontier edge of a statement that defines a, so we must ensure that s_{dst} is a ϕ-node for a and therefore call MAKEPHI. The condition $(s_{dst}, a) \in P_\star$ captures the case where s_{dst} is already a ϕ-node for a, but now there is a new reachable incoming edge, which is handled by lines 4–8 in MAKEPHI as illustrated in Figure 2. Notice that we carefully ensure in FORWARD, MAKEPHI, and CONTINUE that no dataflow is propagated across control-flow edges, in particular frontier edges, until they are known to be reachable.

Proposition *The sparse framework has same analysis precision as the basic framework. Specifically, if $O(s, a) = v$ for some $s \in S$, $a \in A$, and $v \in V$ after analyzing a given program with the sparse framework, then we also have $O(s, a) = v$ when analyzing the program with the basic framework.*

3.4 A Data Structure for Finding Dominating Definitions

During the fixpoint computation new definition sites and use sites are discovered incrementally by the READ and WRITE operations. A key challenge is how to ensure that the FINDDEF operation is able to quickly find the nearest dominating definition for any statement in the control-flow graph.

The naive version of FINDDEF from Section 3.1 is easy to implement, as also suggested by Chase et al. [2]. It is, however, impractical because each invocation requires a traversal along a spine of the dominator tree, in the worst case from the given statement all the way to the root. As an example, consider a straight-line program consisting of k statements in sequence. Invoking FINDDEF at the last statement may then require traversal of all k statements to find the nearest dominating definition.

A better approach is to maintain dominator information separately for each address and only for the nodes that are known to be definition sites or ϕ-nodes. The idea is to equip each statement in the control-flow graph with two numbers,

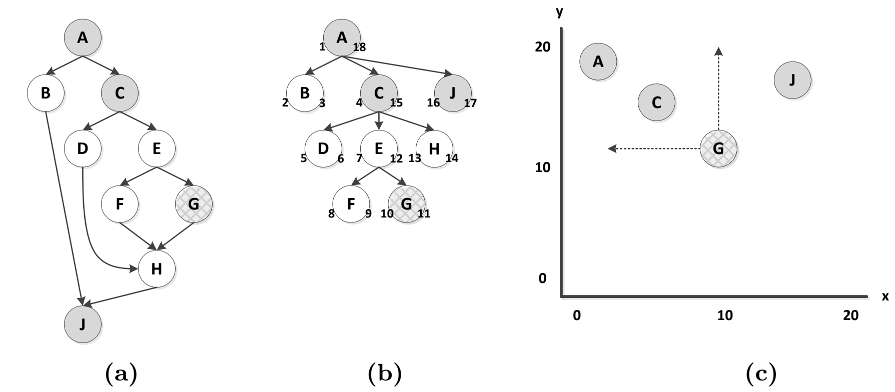

Fig. 3. (a) A control-flow graph fragment where the statements A, C and J are definition sites for some address. The statement G is a use site of the same address. **(b)** The dominator tree for the control-flow graph with the definition sites and use site marked. **(c)** The 2d points associated with the timestamps of the dominator tree.

x and y, as shown in Figure 3. The numbers are obtained through a depth-first traversal of the dominator tree, such that the first number x is the discovery time and the second number y is the exit time. Using these numbers we can determine dominance between nodes: if p dominates q, then p must have been discovered before q, i.e. $x_p < x_q$, and all children of q must have been visited before exiting p, i.e. $y_q < y_p$. As an example, in Figure 3, statement E dominates F since $x_\mathsf{E} < x_\mathsf{F}$ and $y_\mathsf{F} < y_\mathsf{E}$.

A key observation is that to find the *nearest* dominating definition of a node q we need to find a node p where $x_p < x_q$ and $y_q < y_p$. Of all nodes that satisfy these conditions we wish to find the one with maximum x value. For example, in Figure 3 the nearest dominating definition at G is C, which satisfies these properties.

One approach is to store the definitions in a resizable array and perform a linear scan to find the nearest dominating definition, as in Staiger-Stöhr [19]. Unfortunately, this requires $\mathcal{O}(d)$ time, where d is the number of definitions. Another approach is to define an ordering such that finding all dominating definitions takes $\mathcal{O}(\log d)$ time and then scan through these to find the nearest dominating definition, as in Tok et al. [20]. However, the scanning may still require $\mathcal{O}(d)$ time.

Our solution works as follows. If we interpret the number pair (x_q, y_q) of a node q as a point in a two dimensional space, then finding the nearest dominating definition p is equivalent to finding the point (x_p, y_p) in the rectangle $[0, x_q] \times [y_q, \infty]$ with the maximum x_p. This is a well-known problem in the computational geometry literature; one data structure solving this problem is the quadtree [5]. Finding the nearest dominating definition then takes $\mathcal{O}(\sqrt{n})$ time where n is the number of control-flow graph nodes, and a new node can be inserted in $\mathcal{O}(\log n)$ time. Quadtrees are simple to implement and have a low constant-factor overhead. Our experimental comparison (see Section 4) confirms that quadtrees lead to a faster implementation than the naive version of FindDef and Staiger-Stöhr's approach. In principle one could combine the techniques and get $\mathcal{O}(\min(\sqrt{n}, d))$, but in practice using the quadtrees alone seems to work well.

To summarize, we pre-compute the two numbers for every statement in the control-flow graph and then maintain a quadtree y_a for each $a \in A$ containing every statement $s \in S$ where $(s, a) \in O_\star \cup P_\star$. The FindDef procedure from Section 3.1 is then replaced by a search in y_a.

4 Implementation and Evaluation

We have implemented a dataflow analysis for JavaScript, configurable for both the traditional dense propagation (Section 2) and the sparse analysis with on-the-fly SSA construction (Section 3). The dataflow lattices and transfer functions are designed in the style of the TAJS analysis by Jensen et al. [9], structured such that all transfer functions are expressed using the Read, Write and Continue operations. For the quadtrees, we use a variant called *compressed quadtrees* [6]. Interprocedural dataflow is handled by straightforward generalizations of the algorithms from the preceding sections. Call graphs are built on-the-fly, similar to TAJS. For the interprocedural sparse analysis, ϕ-nodes are made at function entries and at no-op statements that are placed after call sites where dataflow may merge from different functions. Searching for dominating definitions may then span multiple functions backward via the call edges, and similarly, definitions inside a function are propagated forward along dominance frontiers of the call sites. The full implementation is approximately 20,000 lines of Scala code, whereof the core that corresponds to Section 2 and 3 constitutes less than 1,000 lines.

Our experiments are based on the collection of JavaScript programs shown in Table 1. (Our current implementation does not contains models of the browser API and HTML DOM, so we settle for stand-alone JavaScript programs.) The collection contains programs from the Mozilla SunSpider and Google Octane benchmark suites, plus a few additional programs found on the web. All experiments are performed on an Intel Core 2 Duo 2.5 GHz PC. The analysis implementation and all benchmarks are available online.[1]

We consider the following three research questions:

Q1: Is our sparse analysis technique more efficient than the basic analysis framework? The literature shows that sparse analysis is usually highly effective, but since none of the existing techniques are applicable to our setting, which involves both pointers and reachability, we cannot know *a priori* whether our sparse analysis has similar advantages.

Q2: How does the performance of our sparse analysis algorithm compare to staged analysis techniques? As argued in Section 1, performing sparse analysis on the basis of imprecise reachability information would affect not only

[1] `http://www.brics.dk/sparse/`

Program			*Total time*		*SSA overhead*			
Name	*Lines*	*Nodes*	*Basic*	*Sparse*	%	*Quadtree*	*Naive*	*Array*
`deltablue.js`	885	3303	*timeout*	35780	43%	15223	23347	21927
`richards.js`	541	1655	*timeout*	703	31%	216	391	405
`splay.js`	398	1058	79844	705	28%	198	268	273
`3d-cube.js`	343	2875	*timeout*	1974	24%	482	991	899
`3d-raytrace.js`	443	3000	*timeout*	2723	30%	812	1686	2954
`access-nbody.js`	170	847	63864	488	39%	189	262	218
`crypto-aes.js`	426	2581	*timeout*	713	25%	177	451	486
`crypto-md5.js`	295	1508	30561	2091	3%	53	111	117
`garbochess.js`	2812	16146	*timeout*	5764	26%	1501	3138	4362
`simplex.js`	450	2121	*timeout*	465	28%	128	282	235
`jpg.js`	889	5146	*timeout*	2621	21%	538	1035	992
`javap.js`	1430	5561	*timeout*	1693	33%	559	1479	3037

Table 1. Experimental results. *Lines* shows the number of source code lines and *Nodes* shows the number of control-flow graph statements for each program, to indicate their sizes. *Basic* and *Sparse* are the total analysis time for the basic and sparse frameworks, respectively. *SSA overhead* shows the time spent on SSA construction during the sparse analysis, using three different implementations for maintaining dominating definitions. All times are shown in milliseconds, with *timeout* representing a timeout of 90 seconds.

the degree of sparseness but also the precision of the main analysis, and we want our sparse analysis to be as precise as with the basic framework. On the other hand, our algorithm could potentially be simplified without affecting analysis precision by using a pre-analysis to compute definition sites and use sites for SSA construction, instead of performing it all on-the-fly. For this reason, it is interesting to measure the overhead of computing that information in our on-the-fly sparse analysis framework.

Q3: Are quadtrees a suitable choice in practice, compared to other techniques for maintaining information about reaching definitions? In Section 3.4 we argued that quadtrees have a good theoretical complexity, however, this needs to be supported empirically.

To answer Q1 we instantiate the analysis with both configurations (using quadtrees for the sparse analysis). The columns *Basic* and *Sparse* in Table 1 show the corresponding analysis times. The numbers show that our sparse analysis is in most cases more than an order of magnitude faster than the basic framework. As result, we have demonstrated that it is possible to perform efficient sparse analysis in a setting that involves pointers and reachability.

We address Q2 by assuming an *ideal* pre-analysis that computes the definition sites and use sites and from this constructs the SSA form, with the full precision of the on-the-fly sparse analysis. Designing a realistic pre-analysis involves a trade-off: it has to be fast (at least, faster than the original dense analysis), and it has to be reasonably precise (since imprecision can lead to less sparseness in the main analysis). With such a pre-analysis, we can perform sparse analysis and still account for reachability – reminiscent of the sparse conditional constant analysis by Wegman and Zadeck [21]. The *SSA overhead* columns (% and *Quadtree*) in Table 1 show how much time is spent by our sparse analysis inside the operations FindDef, Update, Forward, and MakePhi (excluding Propagate), relative to the entire sparse analysis and in milliseconds, when using the quadtree implementation of FindDef. This constitutes work that in principle could be omitted if using a pre-analysis. We observe that between 3% and 43% of the analysis time is spent in these parts. In other words, the best imaginable pre-analysis will only be able to achieve a speedup of less than 1.7x (for `deltablue.js`) and on average less than 1.4x. Moreover, by using our on-the-fly approach, the analysis developer is relieved of the burden of designing and implementing a fast and precise pre-analysis.

Regarding Q3, the columns *Quadtree*, *Naive*, and *Array* show the time for SSA construction with different implementations of the data structure used for finding dominating definitions. The *Quadtree* column corresponds to our quadtree-based implementation described in Section 3.4, *Naive* corresponds to FindDef from Section 3.1, and *Array* follows the approach of Staiger-Stöhr [19], as discussed in Section 3.4. In all cases the quadtree implementation is the fastest and typically outperforms the alternatives by a factor of 1.4x to 5.4x.

5 Related Work

The basic ideas in sparse analysis originate from Reif and Lewis [15] who suggested the use of *global value graphs*, for example for efficient constant propagation analysis. The concept of SSA form is attributed to Rosen et al. [16]. Cytron et al. [4] introduced the concept of dominance frontiers as an effective mechanism for placing ϕ-nodes. As mentioned in the introduction, the *sparse conditional constant* analysis by Wegman and Zadeck [21] builds on top of this work and takes reachability into account during the analysis by tracking which def-use edges represent executable flow. The notion of *dependence flow graphs* by Johnson et al. [11] is a variant of SSA that incorporates branch conditions and thereby supports subsequent dataflow analysis with reachability. Common to this line of work is that heap objects and pointers are not supported.

For programming languages with pointers, most work on sparse analysis has focused on pointer analysis, not dataflow analysis in general. The *semi-sparse* pointer analysis by Hardekopf and Lin [7] uses SSA and sparse analysis for top-level variables that are not accessed via pointers, whereas address-taken variables and heap allocated data are treated using standard flow-sensitive analysis without sparseness.

Other techniques handle pointers typically by staging the analysis using a pre-analysis to approximate possible definition sites and use sites [3, 8, 14]. However, as discussed previously, that approach cannot support reachability without sacrificing analysis precision or sparseness. By computing definition sites and use sites on-the-fly, we avoid that problem.

The analysis by Chase et al. [2] handles pointers and performs sparse analysis on the basis of ϕ-nodes that are computed on-the-fly, however, it does not account for reachability. The analysis framework by Tok et al. [20] is based on similar ideas. The algorithms used in those analyses for finding dominating definitions are discussed in Section 3.4. A related analysis framework has been presented by Staiger-Stöhr [19].

Numerous other program analysis techniques have been designed to prevent various kinds of redundancy in the dataflow propagation. Of particular relevance is the *lazy propagation* technique by Jensen et al. [10] that restricts dataflow at call sites that is not needed by the function being called. When use sites are incrementally discovered, the relevant values are recovered by a backward traversal of the call graph, which is reminiscent of the search for nearest dominating definitions in our sparse analysis. We conjecture that our sparse analysis may be more efficient than lazy propagation; however, lazy propagation is known to work smoothly together with recency abstraction [1,9], which is a useful technique for boosting analysis precision, and it is an open problem whether sparse analysis and recency abstraction can also be combined effectively.

In summary, the present work can be understood as a generalization and combination of ideas from on-the-fly SSA construction [2, 19, 20] while taking reachability into account [11, 21]. Furthermore, we propose a more efficient data structure, based on insights from computational geometry [5], for managing dominating definitions, compared to the existing techniques [19, 20].

6 Conclusion

We conclude that it is possible to perform efficient sparse dataflow analysis in a setting that requires reasoning about pointers and reachability. Our experimental evaluation shows not only that the sparse analysis is significantly faster than the dense counterpart, but also that the overhead of on-the-fly SSA construction is small, which makes the approach a promising alternative to staged analyses. Moreover, we have demonstrated that quadtrees are suitable for maintaining information about dominating definitions.

Our next step is to integrate the analysis algorithm into the TAJS tool to become able to explore the performance on a larger class of JavaScript application. It may also be interesting to apply the algorithm to other programming languages and other abstract domains.

Acknowledgments The authors thank Casper Kejlberg-Rasmussen, Jesper Sindal Nielsen, and Ondřej Lhoták for inspiring discussions about data structures and dataflow analysis. The work presented in this paper was supported by the Danish Research Council for Technology and Production.

References

1. Gogul Balakrishnan and Thomas W. Reps. Recency-abstraction for heap-allocated storage. In *Proc. 13th International Static Analysis Symposium*, August 2006.
2. David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1990.
3. Fred Chow, Sun Chan, Shin-Ming Liu, Raymond Lo, and Mark Streich. Effective representation of aliases and indirect memory operations in SSA form. In *Proc. 6th International Conference on Compiler Construction*, April 1996.
4. Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.
5. Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer, 1997.
6. Sariel Har-Peled. *Geometric Approximation Algorithms*. American Mathematical Society, Boston, MA, USA, 2011.
7. Ben Hardekopf and Calvin Lin. Semi-sparse flow-sensitive pointer analysis. In *Proc. 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 2009.
8. Ben Hardekopf and Calvin Lin. Flow-sensitive pointer analysis for millions of lines of code. In *Proc. 9th International Symposium on Code Generation and Optimization*, April 2011.
9. Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for JavaScript. In *Proc. 16th International Static Analysis Symposium*, volume 5673 of *LNCS*, August 2009.
10. Simon Holm Jensen, Anders Møller, and Peter Thiemann. Interprocedural analysis with lazy propagation. In *Proc. 17th International Static Analysis Symposium*, volume 6337 of *LNCS*, September 2010.
11. Richard Johnson and Keshav Pingali. Dependence-based program analysis. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1993.
12. John B. Kam and Jeffrey D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:305–317, 1977.
13. Gary A. Kildall. A unified approach to global program optimization. In *Proc. ACM Symposium on Principles of Programming Languages*, October 1973.
14. Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, and Kwangkeun Yi. Design and implementation of sparse global analyses for C-like languages. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2012.
15. John H. Reif and Harry R. Lewis. Symbolic evaluation and the global value graph. In *Proc. 4th ACM Symposium on Principles of Programming Languages*, January 1977.
16. Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Global value numbers and redundant computations. In *Proc. 15th ACM Symposium on Principles of Programming Languages*, January 1988.
17. Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. Dynamic determinacy analysis. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2013.

18. Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well: understanding object-sensitivity. In *Proc. 38th ACM Symposium on Principles of Programming Languages*, January 2011.
19. Stefan Staiger-Stöhr. Practical integrated analysis of pointers, dataflow and control flow. *ACM Transactions on Programming Languages and Systems*, 35(1):5:1–5:48, 2013.
20. Teck Bok Tok, Samuel Z. Guyer, and Calvin Lin. Efficient flow-sensitive interprocedural data-flow analysis in the presence of pointers. In *Proc. 15th International Conference on Compiler Construction*, March 2006.
21. Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, 1991.