Lifting On-Demand Analysis to Higher-Order Languages

Daniel Schoepe[1(✉)], David Seekatz[2], Ilina Stoilkovska[1], Sandro Stucki[5], Daniel Tattersall[6], Pauline Bolignano[1], Franco Raimondi[1,3], and Bor-Yuh Evan Chang[4,7]

[1] Amazon, London, UK
{schoeped,ilinas,pln,frai}@amazon.com
[2] Calgary, Canada
[3] Middlesex University, London, UK
[4] University of Colorado Boulder, Boulder, USA
[5] Amazon, Gothenburg, Sweden
satucki@amazon.com
[6] Amazon, Seattle, USA
dtatters@amazon.com
[7] Amazon, Boulder, USA
byec@amazon.com

Abstract. In this paper, we present an approach to lift on-demand analysis to higher-order languages. Specifically, our approach bootstraps an *on-demand call graph* construction by leveraging a pair of on-demand data flow analyses. Static analysis is increasingly applied to find subtle bugs or prove deep properties in large, industrial code bases. To effectively do this at scale, analyzers need to both resolve function calls in a precise manner (i.e., construct a precise call graph) and examine only the relevant portion of the program (i.e., be on-demand). A strawman strategy to this problem is to use fast, approximate, whole-program call graph construction algorithms. However, this strategy is generally not adequate for modern languages like JavaScript that rely heavily on higher-order features, such as callbacks and closures, where scalable approximations often introduce unacceptable imprecision. This strategy also limits increasingly sophisticated *on-demand analyses*, which scale by analyzing only parts of a program as needed: the scalability advantages of an on-demand analysis may be thwarted by the need to construct a whole-program call graph. The key insight of this paper is that existing on-demand data flow analyses can themselves be applied in a black-box manner to construct call graphs on demand. We propose a soundness condition for the existing on-demand analyses with respect to partial call graphs, formalize our algorithm as an abstract domain combinator, and prove it sound in Isabelle/HOL. Furthermore, we evaluate a prototype implementation of the resulting on-demand call graph construction algorithm for a subset of JavaScript (using the Synchronized Push-Down Systems framework as the underlying data flow analysis) on benchmarks making heavy use of higher-order functions.

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2023
M. V. Hermenegildo and J. F. Morales (Eds.): SAS 2023, LNCS 14284, pp. 460–484, 2023.
https://doi.org/10.1007/978-3-031-44245-2_20

1 Introduction

We consider the problem of lifting on-demand static analyses to higher-order languages—that is, transforming, in a sound manner, an on-demand static analysis relying on an upfront call graph into a fully on-demand analysis constructing its own call graph, even in the presence of first-class functions.

Program analysis approaches are becoming more and more sophisticated, increasingly able to find subtle bugs or prove deep program properties of interest in large code bases [9,26]. There are two key enablers for such advances, especially needed to scale to large industrial applications. One is the ability to reason interprocedurally about the behavior across different functions and modules of a program in a precise manner (rather than, e.g., relying solely on local, intraprocedural information or coarse-grained global information such as types). The other is the capabilty to be on-demand (i.e., to examine only the relevant portion of a program to derive a desired fact on demand).

Reasoning interprocedurally requires access to a *call graph* linking call sites in the program to functions that they may invoke at run time. To apply a static analysis interprocedurally, many tools assume that a call graph is provided upfront, and is consulted by the analysis to determine which parts of the program should be explored. This creates two limitations. First, for higher-order, imperative languages such as JavaScript, the combination of first-class functions with a dynamic heap and object-oriented features may require a deep interleaving between call graph construction and data flow analysis. This arises due to the need to precisely track functions as they flow from the points where they are referenced through higher-order functions, heap cells, inheritance hierarchies, and closure bindings. Without this back-and-forth between call graph construction and data flow analysis, precision might be limited or come at the price of soundness or performance trade-offs. Second, this reliance on an upfront call graph limits the benefit of on-demand techniques—a precise data flow analysis to compute a call graph upfront may significantly negate the benefits of a subsequent on-demand analysis. For example, Stein et al. [32] lift an arbitrary abstract interpretation to be on-demand (and incremental) but still assume an upfront call graph.

The key insight of this work is that existing on-demand intraprocedural data flow analyses can *themselves* be leveraged in a black-box manner to bootstrap an on-demand construction of the call graph. The approach starts from an empty call graph and proceeds by interleaving *backward* data flow queries, resolving which values may flow to a given expression, and *forward* data flow queries, resolving which expressions a given value may flow to. Appropriately interleaving such queries allows us to bootstrap a sound overapproximation of a relevant part

of the call graph. This technique allows us to automatically lift the results of on-demand analysis for first-order languages to higher-order ones, thereby further reducing the need for whole-program analysis. As a result, we can parametrically leverage progress on analysis of other challenging language features, allowing the on-demand call graph construction to benefit from the large body of work that already exists on analyzing various combinations of language features, including mutability. Concretely, we make the following contributions:

– We propose a language-agnostic construction for bootstrapping an on-demand call graph, parameterized by a pair of underlying backward and forward on-demand data flow analyses. The two analyses are treated as black boxes, except for the assumption that they can resolve backward and forward queries about data flows between values and expressions with respect to a partial call graph (Sect. 2).
– We present a formalization of our approach as an abstract domain combinator and determine sufficient assumptions on the input analysis and target language to guarantee soundness and termination (Sect. 3). To express soundness, we also introduce a notion of soundness up to a given call graph. This demonstrates a broader approach to formulating and proving soundness of on-demand analyses. Our theoretical results are mechanized in Isabelle/HOL [23]. The theory files are available online [27].
– We evaluate our technique on a prototype implementation that instantiates the approach for a subset of JavaScript, leveraging the intermediate representation of the JavaScript program analyzer TAJS [17], and using Synchronized Push-Down Systems (SPDS) [30] as the underlying data flow analyses. For our evaluation, we use a benchmark set of programs generated via property-based testing techniques, implemented using QuickCheck [5] (Sect. 4). Our results provide some evidence that on-demand call graph construction introduces time savings and explores a smaller portion of the program, when compared with whole-program call graph construction.

2 Overview of Our Approach

In this section, we give an informal overview of our approach to on-demand call graph construction, illustrating the main ideas on a small JavaScript example program (Fig. 1). The presentation in this section is intentionally kept high-level: we assume we are given a forward and a backward data flow analysis that can resolve queries with the help of an existing call graph, but we gloss over the details of how such queries are issued and the formal requirements to make our construction sound. These details are formalized in Sect. 3.

JavaScript programs frequently use callbacks, e.g., to handle user events or interactions between different components in UI frameworks, such as React [21]. Consider the JavaScript snippet in Fig. 1. The function `process` takes two callback arguments: it retrieves data by calling the callback `getData` and passes the result to the callback `handle`. Unfortunately, callbacks complicate the control-flow of programs, which makes them harder to reason about and increases the

```
1  function writeToLog(arg) {
2    log(arg);
3  }
4  function readUserData() {
5    // placeholder for a private
         source
6    return "private userData";
7  }
8  function process(getData, handler) {
9    var data = getData();
10   handler(data);
11 }
12 var handler = writeToLog;
13 process(readUserData, handler);
```

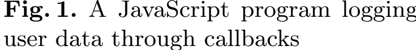


Fig. 1. A JavaScript program logging user data through callbacks

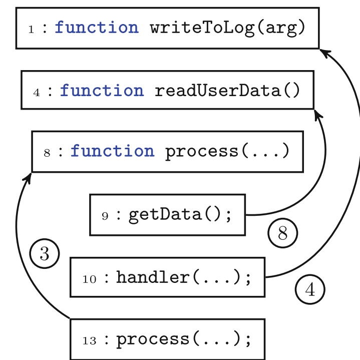


Fig. 2. The call graph constructed for the example program

Query	Step 0	1	2	3	4	5
⟨2, arg, ←⟩	∅	∅	∅	∅	∅	∅
↳⟨1, writeToLog, →⟩		∅	∅	∅	{handler@10}	{handler@10}
↳⟨13, process, ←⟩			∅	{process@8}	{process@8}	{process@8}
↳⟨9, getData, ←⟩						∅
↳⟨8, process, →⟩						

Query	Step 6	7	8	9
⟨2, arg, ←⟩	∅	∅	∅	{private@6}
↳⟨1, writeToLog, →⟩	{handler@10}	{handler@10}	{handler@10}	{handler@10}
↳⟨13, process, ←⟩	{process@8}	{process@8}	{process@8}	{process@8}
↳⟨9, getData, ←⟩	∅	∅	{readUserData@4}	{readUserData@4}
↳⟨8, process, →⟩	∅	{process@13}	{process@13}	{process@13}

Fig. 3. Step-by-step on-demand call graph construction

risk of introducing unintended and undesired behavior. Returning to our example program in Fig. 1, which logs sensitive user data. The leak is not immediately visible since it happens indirectly: `process` is invoked with the arguments `readUserData`, a function returning sensitive data (line 6), and `writeToLog`, a function writing its argument to a public sink (`log` on line 2). Thus, through a sequence of callbacks, the program leaks sensitive user data into a log.

Many existing analyses (e.g., [30]) can detect such leaks, but they typically require a call graph to track interprocedural flows. For example, in order to determine whether `arg` on line 2 might contain sensitive information, an analysis needs to identify all possible calls to `writeToLog`, including indirect ones, such as the call to `handler` on line 10. Constructing a call graph for programs involving callbacks is challenging, particularly for large code bases.

We now describe a construction that lazily computes only those parts of the call graph that are required by a client analysis. This construction relies on two components: (1) a backward data flow analysis that can track which values flow to a given expression, and (2) a forward data flow analysis that determines to which expressions a given value may flow to. Each data flow analysis is only assumed to handle interprocedural flows soundly up to a given call graph. Starting from an empty call graph, we can lazily compute call edges requested by a client analysis by repeatedly issuing queries to the two data flow analyses. We show how this technique applies to the example from Fig. 1; the resulting call graph is shown in Fig. 2. Note that the approach is not limited to tracking information leaks but can be applied to any client analysis requiring call graph information. Furthermore, two different analyses can be used for forward and backward queries as they only communicate through the call graph.

The table in Fig. 3 details the individual steps (1–9) needed to construct the call graph in Fig. 2: each edge in the call graph is labeled by the step number at which it is introduced. The table also tracks the queries issued to the underlying analysis during the process: *backward queries* of the form $\langle \ell, c, \leftarrow \rangle$ to determine which functions can flow to expression c on line ℓ, and *forward queries* of the form $\langle \ell, f, \rightarrow \rangle$ to determine which expressions a function f may flow to. An empty cell in the table indicates that a query has not been issued yet, while $\emptyset$ indicates that a query has been issued but has not yet produced any results. For each step, the table shows how the data flow analyses can make progress given the call graph computed up to this point. This call graph is derived in each step from the answers to queries found so far.

In addition, the table in Fig. 3 also shows the dependency between queries: the "↳" symbol before a query indicates that its result is required to solve an earlier query (i.e., one higher up in the dependency tree). For readability, the table is split into two segments, and should be read following the **Step** number.

Starting from the call to `log` inside function `writeToLog` (on line 2), the client analysis wants to determine whether `arg` may contain sensitive data. Hence, it issues a backward query $\langle 2, \texttt{arg}, \leftarrow \rangle$ to determine which values flow to `arg` (step 0). Because `arg` is an argument to `writeToLog`, this in turn requires identifying call sites of `writeToLog`, for which a forward query $\langle 1, \texttt{writeToLog}, \rightarrow \rangle$ to identify call sites of `writeToLog` (step 1) is issued.

To answer forward queries about calls to a function f, the algorithm starts by finding syntactic references to f in the code and following the flow of f forwards from those points; such references can be found cheaply by analyzing the scope of variables in the program. In this case, the only such reference occurs in the top-level code, where `writeToLog` is assigned to `handler`, which is passed to `process`. To proceed in identifying call sites of `writeToLog`, the analysis needs to find out how the `handler` argument is used inside call targets. To do so, a query $\langle 13, \texttt{process}, \leftarrow \rangle$ is issued to resolve call targets of the call to `process` on line 13 (step 2). Since this is a direct call, the underlying backward data flow analysis resolves the possible call targets to the function `process` defined on line 8 (step 3); this also adds a call edge to our call graph and allows $\langle 1, \texttt{writeToLog}, \rightarrow \rangle$

to make progress by analyzing the body of `process` to find invocations of its `handler` argument. The data flow analysis then finds a call to `handler` in the body of `process` intraprocedurally, so a call edge is added from that expression to `writeToLog` (step 4). This in turn enables further progress on $\langle 2, \texttt{arg}, \leftarrow \rangle$ which can proceed backwards from this call to resolve what values flow to variable `data`, in this case discovering that the value of `getData` is assigned to `data`. Since `getData` is invoked as a function, another query $\langle 9, \texttt{getData}, \leftarrow \rangle$ needs to be issued to resolve callees of `getData` (step 5).

Because `getData` is passed as a parameter to `process`, further interprocedural analysis is needed to make progress and issue a forward query $\langle 8, \texttt{process}, \rightarrow \rangle$ to find call sites of `process` (step 6). The partial call graph already contains a call edge for `process`, but since this is the first forward query for `process`, there might be additional calls not yet discovered; in this case, however, `process` is only referenced by that call and no additional edges are found (step 7).

This call edge allows the query $\langle 9, \texttt{getData}, \leftarrow \rangle$ to make progress by determining that `getData` points to `readUserData` resulting in a new call edge from `getData()` to `readUserData` (step 8). This in turn triggers further progress of $\langle 2, \texttt{arg}, \leftarrow \rangle$ through the newly added call edge into the body of `readUserData` allowing the data flow analysis to discover that `arg` may contain data from a private source (shown as `private`@6 in step 9). This completes the analysis.

The above process is fully on-demand without requiring an analysis of the entire program, aside from identifying syntactic references to functions (which can be determined cheaply through scoping). Additionally, the underlying data flow analyses are reused in a black-box fashion, as long as they allow resolving forward and backward queries up to a given call graph. Rerunning queries when new call edges are added can be avoided when the data flow analyses are incremental and can take newly discovered call edges into account without starting from scratch. Also note that queries make progress independently based on newly discovered call graph edges, rather than requiring to fully resolve each subquery before continuing with a parent query.

2.1 Precision

Different data flow analyses choose different trade-offs in terms of precision and scalability along various dimensions such as context-sensitivity [19], path-sensitivity [8], and how aliasing is handled [10]. For example consider the program in Fig. 4 making use of the JavaScript heap.

A client analysis may need to resolve the calls on lines 15 and 18. Following the algorithm outlined above, this would eventually result in trying to resolve what parameter `f` points to on line 2. A simple data flow analysis that does not track calling contexts may not distinguish between the objects whose `'func'` property is being assigned to in the body of `storeFunc` and therefore report both `function f()` and `function g()` as potential allocation sites of `retrieveFunc(objN)`, resulting in an over-approximate call graph. However, if a calling-context-sensitive analysis is used to resolve call expressions to call tar-

```
1  function storeFunc(obj, f) {
2      obj['func'] = f;
3  }
4  function retrieveFunc(obj) {
5      return obj['func'];
6  }
7  function f() {
8      // ...
9  }
10 function g() {
11     // ...
12 }
13 var obj1 = {};
14 storeFunc(obj1, f);
15 retrieveFunc(obj1)();
16 var obj2 = {};
17 storeFunc(obj2, g);
18 retrieveFunc(obj2)();
```

Fig. 4. JavaScript program with heap use

gets, the query $\langle 15, \texttt{retrieveFunc(obj1)}, \leftarrow\rangle$ returns `function f()` as the only callee, and $\langle 18, \texttt{retrieveFunc(obj2)}, \leftarrow\rangle$ is resolved to `function g()` only.

To see why this is the case, consider the subqueries created during analysis: The initial query $\langle 15, \texttt{retrieveFunc(obj1)}, \leftarrow\rangle$ starts with an empty calling context, the body of `retrieveFunc(obj1)` needs to be analyzed, leading to query $\langle 15, \texttt{retrieveFunc}, \leftarrow\rangle$ to identify the callee, which immediately yields `function retrieveFunc()`. Its analysis in turn yields the fact that the query result is the functions that may flow to `obj1['func']`. In order to analyze how `obj1` is modified by `storeFunc`, another query to resolve the call target on line 14 is needed, returning `function storeFunc()` immediately, adding a call edge to the call graph. This allows the initial query $\langle 15, \texttt{retrieveFunc(obj1)}, \leftarrow\rangle$ to proceed analyzing the body of `storeFunc`. Since we assumed the backward data flow analysis analysis to be calling-context-sensitive, this analysis will use the call on line 14 as the calling context, allowing it to determine that `f` was stored in `obj1['func']`. The calling context of the underlying analysis is preserved in the analysis of an individual query in the same manner as when the underlying analysis is invoked with a precomputed call graph instead.

The same argument applies regardless of the precision of the context sensitivity or how exactly the analysis represents calling contexts and generalizes to other forms of sensitivity, such as field sensitivity.

In more complex cases involving multiple queries, the constructed call graph may contain spurious edges. For example, consider a version of Fig. 1 where the top-level code makes two calls: `process(readUserData, dontWriteToLog)` and `process(readPublicData, writeToLog)`, such that `readPublicData` returns public information and `dontWriteToLog` does not log its argument. In this case, the call on line 10 is (correctly) resolved to call targets `dontWriteToLog` and `writeToLog`, and the call 9 is (correctly) resolved to call targets `readPublicData` and `readUserData`. A standard data flow analysis, when given this call graph, will report a warning when the flow is terminated in the body of `readUserData`. This problem can be addressed by augmenting queries with a context parameter specific to the underlying data flow analyses. In this case, when resolving backward query $\langle 2, \texttt{arg}, \leftarrow\rangle$, additional calling context could be passed to query $\langle 9, \texttt{getData}, \leftarrow\rangle$, eliminating the false positive, while making the analysis more costly as queries now may have to be resolved multiple times with different contexts. We leave such an extension as future work.

2.2 Termination and Soundness

Given that issuing a query may in return issue subqueries, cyclical query dependencies may arise during analysis. The algorithm avoids non-termination due to query cycles by not blocking on a subquery to complete before proceeding. Instead, the algorithm allows each query to make progress whenever any other query finds additional call edges relevant to another query. More precisely, if a query q_1 is processed, which triggers another query q_2 to be issued, and if q_2 in turn again issues q_1, the algorithm discovers that q_1 has already been issued and proceeds with other analysis paths of q_2 that do not depend on q_1 (if there are any). Whenever q_2 finds a call edge, this may allow q_1 to make further progress, in turn possibly allowing further progress of q_2. This process is guaranteed to terminate since, at each step, either the number of query results or the size of the call graph increases, yet both are ultimately bounded by the size of the program. The process is reminiscent of a Datalog-based analysis such as Doop [29] where new tuples being discovered for one relation, may trigger additional rules to apply for another relation, possibly in a mutually recursive fashion. More generally, it is an instance of a fixpoint computation on a finite domain.

Soundness is less straightforward to establish, as it requires issuing all necessary subqueries to eventually discover all the ways in which a given function may be invoked or to find all values flowing to a given expression. To show soundness, we make the assumption that the program under analysis is *closed*, that is, it neither uses reflection nor has entry points that take function arguments. Informally, to see why the on-demand call graph construction algorithm is sound in the absence of reflection, consider how a given function f might be invoked during execution of a closed program. In order for f to be invoked, at least one of the following must hold: (i) f is an entry point to the program, or (ii) a reference to f flows from f's definition to a call site. In other words, references to a function cannot be "guessed" and can be obtained either through a syntactic occurrence of the function's name or through the use of reflection. By starting from such syntactic references, the analysis can track where this function may flow. Similarly, by starting backwards from a call site and using forward analysis to find calls to the surrounding function, the analysis can discover all function definitions flowing to the call site.

Soundness in the presence of reflection requires a way to soundly identify locations where references to a given function may be obtained, for example using an existing reflection analysis [18,20,28]. In practice this may result in significant overapproximation, since analyzing many reflection facilities, for example in Java or JavaScript, requires reasoning about e.g., string computations. We discuss this topic in more detail in Sect. 3.6.

Further, handling programs that are incomplete, that is, programs with missing code such as libraries, is orthogonal to this work. Analyzing incomplete programs with this technique would require incorporating an existing approach for handling to missing code, such as models of missing library functions.

3 On-Demand Call Graph Soundness

In this section, we formally prove the soundness of our approach. We start by introducing a formal, semantic model of programs, call graphs and queries. We define the desired property, *on-demand call graph soundness*, that the call graphs produced by our analysis should satisfy with respect to the given program semantics. We then introduce our algorithm, in the form of a transition system defined by a set of inference rules. Finally, we state our soundness result—that call graphs generated by our rules for a given input program and set of input queries are on-demand sound—and sketch a proof. The full proof has been formalized using Isabelle/HOL and is available online [27].

We choose to depart from the usual abstract interpretation or data flow style presentation in two important ways. First, rather than starting from a concrete program syntax and semantics, we choose to model programs directly as the *collection of (concrete) call-traces* they may produce. We do so because our approach is fundamentally language-agnostic. The call-trace semantics serves as a generic starting point that abstracts over language-specific details while still fitting the abstract interpretation framework. Second, we do not introduce an abstract representation of sets of program traces as it would be used by a realistic analyzer. Instead, we model analyses directly in terms of the (semi-concrete) call-trace semantics. In practice, one does need an abstract domain, and the underlying analysis would typically provide one (e.g., we use SPDS in our prototype implementation). For the purpose of our soundness proof, however, we are primarily interested in the concretization of abstract states back into the concrete call-trace semantics. In our formalization, we therefore skip the indirection through an abstract domain and directly express the analyses as producing sets of concrete traces, while implementations—such as our prototype—use abstract representations that can be queried for various properties that we extract from traces directly in the formal model.

3.1 Program Semantics

Figure 5 defines the language of events in terms of a given set of expressions and functions. In addition, we assume a set of unique *syntactic reference points* where an initial reference to a function is obtained in the program.

$$
\begin{array}{rrl}
\text{call sites } c & \in & \mathbf{CallSite} \\
\text{functions } f & \in & \mathbf{Func} \\
\text{reference points } r & \in & \mathbf{RefPoint} \\
\text{events } e & \in & \mathbf{Event} \\
 & ::= & \mathbf{call}(c, f, r) \\
 & | & \mathbf{enter}(c, f) \\
 & | & \mathbf{exit}(c, f) \\
 & | & \mathbf{return}(c, f) \\
 & | & \mathbf{ref}(f, r) \\
\text{traces } \tau & \in & \mathbf{Event}^* = \mathbf{Trace}
\end{array}
$$

Fig. 5. Syntax of traces and events

Events. Let **CallSite** be the finite set of *call sites* and **Func** the finite set of *function definitions* appearing in some program. Let **RefPoint** be a finite set of *reference points* containing unique identifiers r used to link function calls

```
1 function f() {          // enter(x(), f)
2   return;               // exit(x(), f)
3 }
4 var x = f;              // ref(f, r)
5 x();                    // call(x(), f, r), return(x(), f)
```

Fig. 6. An example program annotated with the events it creates.

to locations where functions are referenced in the source code. To make our approach applicable to a wide range of languages, regardless of their memory model and other features, we model only the language semantics relevant to defining a call graph. To do so, we fix a set **Event** of *events*, where each event $e \in \textbf{Event}$ is one of the following:

- $\textbf{call}(c, f, r)$: A call to function f from call site c, where the reference to f was obtained by the reference point r.
- $\textbf{enter}(c, f)$: Entering the body of callee f, whose call originated from call site c.
- $\textbf{exit}(c, f)$: Exiting the body of callee f, whose call originated from call site c.
- $\textbf{return}(c, f)$: Returning to a call site of function f, whose call originated from call site c.
- $\textbf{ref}(f, r)$: Obtaining a reference to function f with reference point r. This can be either a syntactic reference to f or a use of reflection evaluating to f.

When a function is called, a **call** event is first emitted on the caller side, then an **enter** event is emitted on the callee side. Similarly, when the called function returns, an **exit** event is first created on the callee side, and then a **return** event is created on the caller side. Observe that both the **ref** event and the **call** event contain the reference point r. As we will see in Sect. 3.4, this allows linking the call to a specific reference to the callee that triggered the call. An example program and the created events can be seen in Fig. 6.

Traces and Programs. A *(call) trace* $\tau \in \textbf{Trace}$ is a finite sequence of events. We denote by $|\tau| \in \mathbb{N}$ the length of τ and by τ_i its i-th element, where $0 < i \leq |\tau|$. Given a pair of traces τ, τ', we denote their concatenation by $\tau \cdot \tau'$. We denote by $e_1 \cdots e_n$ the trace consisting of the events $e_1, \ldots, e_n$.

In the following, we fix a program and model its semantics as the (potentially infinite) set of call traces its executions may result in, written $\mathcal{S} \in \mathcal{P}(\textbf{Trace})$. We impose a few restrictions on $\mathcal{S}$ to exclude ill-formed traces that could not be generated by a real program: we assume that events on the caller side are followed by corresponding events on the callee side and vice versa, and that each call to a function f is preceded by obtaining a reference to f. Formally:

- $\mathcal{S}$ is prefix-closed, that is, for every trace $\tau \cdot \tau' \in \mathcal{S}$, the prefix τ is also in $\mathcal{S}$.
- For each trace $\tau \in \mathcal{S}$:
  - If $\tau_i = \textbf{enter}(c, f)$, then $i > 1$ and $\tau_{i-1} = \textbf{call}(c, f, r)$ for $r \in \textbf{RefPoint}$.
  - If $\tau_i = \textbf{call}(c, f, r)$ and $i + 1 \leq |\tau|$, then $\tau_{i+1} = \textbf{enter}(c, f)$.
  - If $\tau = \tau' \cdot \textbf{call}(c, f, r)$, then $\tau \cdot \textbf{enter}(c, f) \in \mathcal{S}$.
  - If $\tau_i = \textbf{exit}(c, f)$, then $\exists j \in \mathbb{N}$, with $0 < j < i$ where $\tau_j = \textbf{enter}(c, f)$

- If $\tau_i = \mathbf{return}(c, f)$, then $i > 1$ and $\tau_{i-1} = \mathbf{exit}(c, f)$.
- If $\tau_i = \mathbf{exit}(c, f)$ and $i + 1 \leq |\tau|$, then $\tau_{i+1} = \mathbf{return}(c, f)$.
- If $\tau_i = \mathbf{call}(c, f, r)$, then $\exists\, j \in \mathbb{N}$, with $0 < j < i$, such that $\tau_j = \mathbf{ref}(f, r)$.

3.2 Queries

A *client analysis* is an analysis that uses a call graph in order to reason about interprocedural control flow in a program. We distinguish two kinds of *call graph queries* that a client analysis can issue:

1. *callee query*, i.e., a call site $c \in \mathbf{CallSite}$, whose purpose is to find all functions $f \in \mathbf{Func}$ that may have be called at c.
2. *caller query*, i.e., a function $f \in \mathbf{Func}$, whose purpose is to find all call sites $c \in \mathbf{CallSite}$ from which the function f may be called.

Intuitively, callee queries are *backward* data flow queries: given a call site c, we want to find all the reference points in the program before c from which function values can flow to c. Conversely, caller queries are *forward* data flow queries: given a function f, we want to find all the reference points in the program from which f can flow to call sites later on in the program.

In each case, the query implicitly defines a subset of program subtraces containing the reference points of interest. Intuitively, these are the parts of program runs relevant to answering a query. Formally, we first define complete backward subtraces with both start and end points and then close the resulting set of subtraces under suffix: $\textit{complete-bwd-traces}(c) = \{\tau \mid \exists \tau_0 . \tau_0 \cdot \tau \in \mathcal{S} \wedge \tau = \mathbf{ref}(f, r) \cdot _ \cdot \mathbf{call}(c, f, r)\}$ and $\textit{bwd-traces}(c) = \{\tau_2 \mid \exists \tau_1 . \tau_1 \cdot \tau_2 \in \textit{complete-bwd-traces}(c)\}$. Note that the reference point r is only used to delimit the backward traces relevant to a given call site c. Similarly, we define the set of *forward subtraces* $\textit{fwd-traces}(f)$ for a forward query f, as the subtraces starting with a reference to f, that is, $\textit{fwd-traces}(f) = \{\tau \mid \exists \tau_0 . \tau_0 \cdot \tau \in \mathcal{S} \wedge \tau = \mathbf{ref}(f, _) \cdot _\}$.

3.3 Call Graphs

To define an on-demand call graph, we first need the notions of a call graph and a whole-program call graph.

A *call graph* $G \subseteq \mathbf{CallSite} \times \mathbf{Func}$ is a directed graph whose vertices are call sites and functions and whose edges connect a call site $c \in \mathbf{CallSite}$ to a function $f \in \mathbf{Func}$. We write $\mathbf{CG} = \mathbf{CallSite} \times \mathbf{Func}$ for the set of all call graphs. We define the *whole-program call graph* of a program, written *whole-cg*, as the set of all pairs (c, f) that occur on a **call** event in some trace of the program semantics. Formally, $\textit{whole-cg} = \{(c, f) \mid \exists \tau \in \mathcal{S}, 0 < i \leq |\tau| \,.\, \tau_i = \mathbf{call}(c, f, _)\}$.

Let $C \subseteq \mathbf{CallSite}$ and $F \subseteq \mathbf{Func}$ be the sets of callee and caller queries, respectively, that a client analysis issues while analyzing the program. Observe that the whole-program call graph *whole-cg* may be large and include many call edges (c, f) that are never used to answer a call graph query, i.e., such that $c \notin C$ or $f \notin F$. The goal of our on-demand approach is to compute a subgraph G of *whole-cg* containing all the edges needed to answer the queries in C and F.

To characterize such on-demand call graphs G, we introduce the notion of (C, F)-*soundness*. Intuitively, a (C, F)-sound on-demand approximation of a whole-program call graph contains at least the call graph edges necessary to answer all client queries. Formally:

Definition 1 (On-demand call graph soundness). *Let* $C \subseteq$ **CallSite** *and* $F \subseteq$ **Func** *be finite sets of callee and caller queries, respectively. A call graph* $G \subseteq$ **CallSite** $\times$ **Func** *is* on-demand call graph sound *w.r.t.* C *and* F *(or simply* (C, F)-sound*) iff every edge* $(c, f) \in$ *whole-cg is in* G *if either* $c \in C$ *or* $f \in F$*.*

3.4 Answering Call Graph Queries

To construct an on-demand call graph, our algorithm starts from an empty call graph, and gradually adds edges based on answers to the callee queries C and caller queries F issued by the client analysis. To find the answers of these queries, our approach is parameterized by two data flow analyses, defined as follows:

- a forward analysis $\mathscr{F} : \mathbf{CG} \times \mathbf{Func} \to \mathcal{P}(\mathbf{Trace})$, used to detect the call sites where a caller query $f \in F$ may have been called; and
- a backward analysis $\mathscr{B} : \mathbf{CG} \times \mathbf{CallSite} \to \mathcal{P}(\mathbf{Trace})$, used to detect the functions that a callee query $c \in C$ may call.

For example, to detect the functions that the callee query $x()$ on line 5 in Fig. 6 may call, our algorithm uses a backward data flow analysis to issue a backward query, whose answer contains the function f, defined on line 1. Once f is obtained as an answer, an edge $(x(), f)$ is added to the on-demand call graph.

To guarantee on-demand soundness of the call graph obtained by applying our algorithm, we assume the following about the underlying data flow analyses $\mathscr{F}$ and $\mathscr{B}$. Both data flow analyses are on-demand analyses, that is, intuitively they need only discover interprocedural data flows between a call site c and function f if the given partial call graph contains the edge (c, f). Their answers are an overapproximation of the set of subtraces relevant to a given call graph query. Note that $\mathscr{F}$ and $\mathscr{B}$ are modeled as functions returning sets of traces in order to reason about the soundness of the approach. In practice they would return abstract representations that concretize to sets of traces, which would provide interfaces to determine when data flows to function boundaries.

Next, we define the notions of backward and forward compatibility of a given trace with a partial call graph. These notions are used to restrict the traces that need to be overapproximated by the on-demand analyses $\mathscr{F}$ and $\mathscr{B}$, since they are allowed to only reason about parts of traces relevant to the given query. A trace τ is *forward-compatible* with a call graph G, written $compat^{\rightarrow}(G, \tau)$, if for any event $\mathbf{enter}(c, f)$ or $\mathbf{return}(c, f)$ in τ, it holds that $(c, f) \in G$. Similarly, a trace is *backward-compatible* with a call graph G, written $compat^{\leftarrow}(G, \tau)$, if for any event $\mathbf{call}(c, f, r)$ or $\mathbf{exit}(c, f)$ in τ, it holds that $(c, f) \in G$. Note that the compatibility definitions are slightly different depending on the direction of the analysis. In the forward case, encountering a call site is easy to identify, but determining the callee-side **enter** event requires resolving which function flows

to the call site. In the backward case, reaching the entry point of a function is easy to identify, but not where this function was called. Also, when proceeding backwards, a call site is indicated first by a caller-side **return** event, but its callee needs to be found through additional analysis to identify the corresponding **exit** event. The precise definitions of $\mathit{compat}^{\rightarrow}(G, \tau)$ and $\mathit{compat}^{\leftarrow}(G, \tau)$ can be found in the Isabelle/HOL formalization [27].

Finally, we define the soundness requirements on $\mathscr{F}$ and $\mathscr{B}$. Given a relevant subtrace that is compatible with a given partial call graph, $\mathscr{F}$ should discover next possible event relevant to a query, whereas $\mathscr{B}$ should discover previous possible events. Intuitively, this captures that analyses can make progress on the part of the program that a partial call graph provides enough information about.

Formally, $\mathscr{F}$ is *forward-sound* iff $\{\tau \in \mathit{fwd\text{-}traces}(f) \mid \mathit{compat}^{\rightarrow}(G, \tau)\} \subseteq \mathscr{F}(G, f)$ and if $\tau \cdot \mathbf{enter}(c, f) \in \mathit{fwd\text{-}traces}(f)$ and $\mathit{compat}^{\rightarrow}(G, \tau)$, then $\tau \cdot \mathbf{enter}(c, f) \in \mathscr{F}(G, f)$. Note that this definition entails that $\mathscr{F}(G, f)$ over-approximates all references $\mathbf{ref}(f, r)$ to f as singleton traces $\mathbf{ref}(f, r)$ are compatible with any call graph. Similarly, $\mathscr{B}$ is *backward-sound* iff for any $\tau \in \mathit{bwd\text{-}traces}(c)$ such that $\tau = \tau' \cdot \mathbf{call}(c, f, r)$ and $\mathit{compat}^{\leftarrow}(G, \tau')$, then $\tau \in \mathscr{B}(G, c)$ and if $\mathbf{call}(c, f, r) \in \mathit{bwd\text{-}traces}(c)$, then $\mathbf{call}(c, f, r) \in \mathscr{B}(G, c)$.

If all the assumptions on $\mathscr{F}$ and $\mathscr{B}$ outlined in this section are satisfied, our call graph construction is sound. We make this precise next.

3.5 On-Demand Call Graph Construction as a Transition System

Our on-demand call graph construction algorithm maintains a state consisting of a triple (G, C, F), where G is the currently known call graph, C contains the set of relevant backward queries for callees of call sites $c \in C$ and F contains a set of relevant forward queries for callers of functions $f \in F$.

Figure 7 describes how call graph construction, starting from some call graph construction state can make progress. If the underlying data flow analyses discover a call for a query, either in the forward or backward direction, then we can add the corresponding call edge to the call graph (rules AddFwdCallEdge and AddBwdCallEdge). Note that discovering a call in the forward direction is indicated by an $\mathbf{enter}(\cdot, \cdot)$ event, whereas in the backward direction, this is indicated by a $\mathbf{call}(\cdot, \cdot, \cdot)$ event instead. This difference results from the fact that when proceeding forwards through a function, the caller-side **call** event is always compatible with any partial call graph, whereas discovering the corresponding callee-side **enter** event must have been resolved by the forward analysis. Similarly, in the backward direction, the **enter** event is backward-compatible with any call graph, while the corresponding **call** event is not. The remaining rules describe which additional queries to issue: When reaching a call in either forward or backward direction, the call target needs to be resolved through an additional query (rules ReachedCallBwds and ReachedCallFwds); these rules again exhibit the same difference regarding **enter** and **call** events as for adding call edges to the call graph. Lastly, when reaching the beginning of a function going backwards or the end of a function going forwards, call sites of the containing function need to be resolved through another query to make progress.

$$\textsc{AddFwdCallEdge}\quad \frac{f \in F \qquad \tau \cdot \textbf{enter}(c,f) \in \mathscr{F}(G,f) \qquad (c,f) \notin G}{(G, C, F) \rightsquigarrow (\{(c,f)\} \cup G, C, F)}$$

$$\textsc{AddBwdCallEdge}\quad \frac{c \in C \qquad \textbf{ref}(f,r) \cdot \tau \cdot \textbf{call}(c,f,r) \in \mathscr{B}(G,c) \qquad (c,f) \notin G}{(G, C, F) \rightsquigarrow (\{(c,f)\} \cup G, C, F)}$$

$$\textsc{ReachedCallFwds}\quad \frac{f \in F \tau \cdot \textbf{call}(c',f',r') \in \mathscr{F}(G,f) \qquad c' \notin C}{(G, C, F) \rightsquigarrow (G, C \cup \{c'\}, F)}$$

$$\textsc{ReachedCallBwds}\quad \frac{c \in C \qquad \textbf{return}(c',f') \cdot \tau \in \mathscr{B}(G,c) \qquad c' \notin C}{(G, C, F) \rightsquigarrow (G, C \cup \{c'\}, F)}$$

$$\textsc{ReachedEnterBwds}\quad \frac{c \in C \qquad \textbf{enter}(c',f') \cdot \tau' \in \mathscr{B}(G,c) \qquad f' \notin F}{(G, C, F) \rightsquigarrow (G, C, F \cup \{f'\})}$$

$$\textsc{ReachedExitFwds}\quad \frac{f \in F \qquad \tau \cdot \textbf{exit}(c,f') \in \mathscr{F}(G,f) \qquad f' \notin F}{(G, C, F) \rightsquigarrow (G, C, F \cup \{f'\})}$$

Fig. 7. On-Demand Call Graph Construction as a transition system

The transition system is intentionally non-deterministic: At each point, multiple rules may be applicable to make progress. The rules can be applied in any order to reach an overapproximation of the relevant parts of the real call graph. We prove soundness of the approach by stating that once the algorithm reaches a fixed point, the call graph is on-demand sound w.r.t. the answered queries.

Theorem 1 (On-Demand Soundness). *For any call graph construction state* (G, C, F)*, if* $(G, C, F) \rightsquigarrow^* (G', C', F') \not\rightsquigarrow$*, then* G' *is* (C', F')*-sound.*

The analysis starts with an empty call graph and non-empty query sets, relying on the special case of the theorem where $G = \emptyset$. As $\rightsquigarrow$ is monotone (discussed below), any queries issued as part of C or F are still present in C' or F'. Based on the intuitions presented in Sect. 2.2, we present a proof sketch summarizing the key techniques. Some definitions and lemma statements are simplified. The full details can be found in the Isabelle/HOL formalization [27].

Proof (sketch). The proof proceeds in four main steps:

1. We first define an intermediate collecting semantics $\rightarrowtail$ that adds subtraces and new queries in the same order as $\rightsquigarrow$ adds call edges and queries. The intermediate collecting semantics maintains the state of forward queries (resp. backward queries) as partial maps $\mathcal{F}$ (resp. $\mathcal{B}$) from functions $f \in \textbf{Func}$ (resp.

$c \in \mathbf{CallSite}$) to subsets of $\mathit{fwd\text{-}traces}(f)$ (resp. $\mathit{bwd\text{-}traces}(c)$). Each step $(\mathcal{F}, \mathcal{B}) \rightarrowtail (\mathcal{F}', \mathcal{B}')$ either adds an event to the end (resp. beginning) of a set in the co-domain of either map. If the event requires no other queries (such as an $\mathbf{ref}(f, r)$ event), then it is added directly. If the new event requires resolving another query (such as function call events), then it is only added if the query it depends on has made enough progress. Alternatively, a step may issue an additional query under similar conditions as $\rightsquigarrow$. Note that this intermediate semantics is more precise as only real events are added to traces. This also renders it not computable.
2. We proceed by proving that $\rightsquigarrow$ overapproximates $\rightarrowtail$. Concretely, we show that if $(\mathcal{F}, \mathcal{B}) \rightarrowtail (\mathcal{F}', \mathcal{B}')$, and $\gamma((G, C, F)) = (\mathcal{F}, \mathcal{B})$, then there exists a new call graph construction state (G', C', F') such that $(G, C, F) \rightsquigarrow (G', C', F')$ and $(\mathcal{F}', \mathcal{B}') \sqsubseteq \gamma((G', C', F'))$, where we write γ for the concretization of a call graph construction state (G, C, F) into subsets of traces for each forward and backward query $(\mathcal{F}, \mathcal{B})$ and $\sqsubseteq$ for a lifting of subset inclusion of traces for each forward and backward query. This is proven using a straightforward induction on $(\mathcal{F}, \mathcal{B}) \rightarrowtail (\mathcal{F}', \mathcal{B}')$.
3. Next, we show that a fixed point of $\rightarrowtail$ approximates all subtraces for the queries that were generated. For this, we need an intermediate definition of well-formedness on the events in the subtraces being discovered. Formally, if $(\mathcal{F}, \mathcal{B}) \not\rightarrowtail$, and $(\mathcal{F}, \mathcal{B})$ is *well-formed*, then for each f such that $\mathcal{F}(f) = T_f$, we have $\mathit{fwd\text{-}traces}(f) \subseteq T_f$. Similarly, if $\mathcal{B}(c) = T_b$, then it holds that $\mathit{bwd\text{-}traces}(c) \subseteq T_b$. Well-formedness for forward queries requires that all singleton traces $[\mathbf{ref}(f, \cdot)]$ are included in T_f and that T_f is prefix-closed. Similarly, well-formed backward sets T_b need to include the singleton suffixes of $\mathit{bwd\text{-}traces}(c)$ in addition to being suffix-closed. To show that a fixed point of $\rightarrowtail$ approximates all subtraces, we proceed by contradiction. Suppose that after reaching a fixed point $(\mathcal{F}, \mathcal{B})$ there is a missing event for a query; hence there must be an earlier missing event. This means there must either be another possible transition $\rightarrowtail$ or an earlier missing event, yielding a contradiction in either case.
4. Combining 2 and 3, we obtain that a fixed point of $\rightsquigarrow$ overapproximates the relevant subset of the whole-program call graph of a given program.

Note that termination follows from the assumption that a fixed program has a finite number of functions and call sites, combined with the monotonicity of $\rightsquigarrow$:

Lemma 1 (Monotonicity). *If* $(G, C, F) \rightsquigarrow (G', C', F')$*, then* $(G, C, F) \sqsubset (G', C', F')$*, where* $\sqsubset$ *denotes lexicographic tuple ordering.*

3.6 Discussion

Reflection. The above algorithm relies on correctly identifying all references to a function f for which call sites need to be determined. In languages without reflection, this can be done easily by identifying where f is referenced syntactically, taking into account scoping rules. However, many languages including Java and JavaScript allow obtaining a reference to a function through the use of

reflection. The above definitions assume that an $\mathscr{F}$ correctly overapproximates where a function might be referenced, which entails a reflection analysis in order to soundly analyze programs in the presence of reflection.

Implementation Considerations. We model $\mathscr{F}$ and $\mathscr{B}$ as returning sets of traces that on-demand call graph constructions inspects for certain events. In a real-world implementation, data flow analyses would provide an interface signaling discovered data flows between reference points and call sites as well to a function boundary. Our prototype, described in Sect. 4, interfaces with the automata-based abstraction of SPDS to detect when to issue additional queries or add call graph edges through the listener functionality provided by SPDS.

Non-termination. In the formal model, non-terminating programs are represented as infinite sets of finite traces. In order for an underlying data flow analysis to be considered sound, such infinite sets need to be over-approximated to satisfy the conditions of forward or backward soundness. In practice, this requires a suitable finite representation of an infinite set of traces. For example, consider the program |while(true) f(); |, producing an infinite sequence of call events $\mathbf{call}(f(), f, r)$ for some reference point r along with associated **enter**, **exit**, and **return** events. Assume further that the call target of `f()` produces no additional events. Its denotation is the infinite set $\{S, S \cdot S, S \cdot S \cdot S, \ldots\}$ where $S = \mathbf{call}(f, f, r) \cdot \mathbf{enter}(f(), f) \cdot \mathbf{exit}(f(), f) \cdot \mathbf{return}(f(), f)$.

To satisfy the conditions of backward soundness, a backward data flow analysis $\mathscr{B}$ has to include any trace S^n in the set $\mathscr{B}(G, f())$. As discussed, a practical implementation will therefore have to finitely represent an infinite set. For example, a backward analysis may map program locations to potential call targets—in this case mapping `f()` to the function f. In SPDS, this example can be represented using a loop in the call push-down system, adding an edge from `f()` to itself. Similar considerations apply to sound forward data flow analyses.

4 Evaluation

The main research question we explore with our experimental evaluation concerns scalability: A key promise of on-demand call graph construction is the application of more expensive analyses to only relevant parts of a code base, rather than the entire program. This unlocks the possibility to apply analyses that are too expensive to use with a whole-program approach.

In addition to a set of initial queries, issued for the call sites of interest in a given program, our on-demand call graph construction issues queries on which the initial queries depend. As a result, how much of a program is explored during analysis depends on the structure of the program and cannot be bounded upfront—in the worst case, the algorithm may still produce the whole-program call graph. Our experiments evaluate how many queries are resolved in total, for initial sets of various sizes, and report on the potential time savings from

on-demand analysis on a set of synthetic benchmarks. The prototype can be found online [27].

Implementation. We implemented the on-demand call graph construction algorithm in a prototype, called MERLIN, for a limited subset of JavaScript. The implementation uses the TAJS [17] intermediate representation and Synchronized Push-Down Systems (SPDS) [30] as the underlying data flow analysis for both forward and backward queries. To support (a subset of) JavaScript in SPDS, the implementation adds backward and forward flow functions on top of an existing language-agnostic SPDS implementation [6]. The implementation supports basic JavaScript features, such as assignments, object allocation and function calls, including closures and accounting for mutability of captured variables. Instantiating SPDS to a sufficiently large subset of JavaScript to analyze real-world code is out of scope for this paper.

To compute a fixed point, MERLIN maintains a set of queries, in addition to the current call graph. A query in this context is represented as a synchronized pushdown system starting from a reference to a function or an call site, depending on the query. Individual queries subscribe to updates about (i) callees of a particular call site, or (ii) call sites of a particular function discovered by other queries. An update may result in adding further transitions in a query's SPDS in the current call graph. Objects are also tracked using SPDS.

When a function entry point is reached by a backward query, or a return statement of a function is reached by a forward query, a new forward query is issued to find call sites of that function. Similarly, when a function call is reached by either a forward or a backward query, a new backward query is issued to resolve possible callees. Based on the results of these new queries, the analysis continues at the function's call sites or a call site's callees.

The asynchronous saturation process described in Sect. 2 is implemented using a reactive programming [12] approach implemented using the Java Virtual Machine's `ForkJoinPool` to resolve queries concurrently. This also enables parallel execution of multiple queries on multi-core machines.

Synthetic Benchmarks. We evaluate MERLIN against a set of synthetic benchmarks generated using the property-based testing library QuickCheck [5]. To capture non-trivial dependencies between the call graph queries, the generated programs heavily use higher-order functions, and treat functions as first class values, reflecting the dynamic nature of JavaScript programs. That is, functions are passed along a chain of functions, both as arguments and return values, before they are eventually called.

Each generated program has between 6000 and 10000 lines of code, including whitespace and braces. Figure 8 shows a (simplified) excerpt from an example in the benchmark set, where function `chainTarget106` is returned from a function via `returnAFunc100`, the return value of which is invoked in `fun57`. The benchmarks contain between 600 and 900 functions, with higher-order call chains of up

to length 4. We leave an investigation of typical usage patterns of higher-order functions in JavaScript for future work.

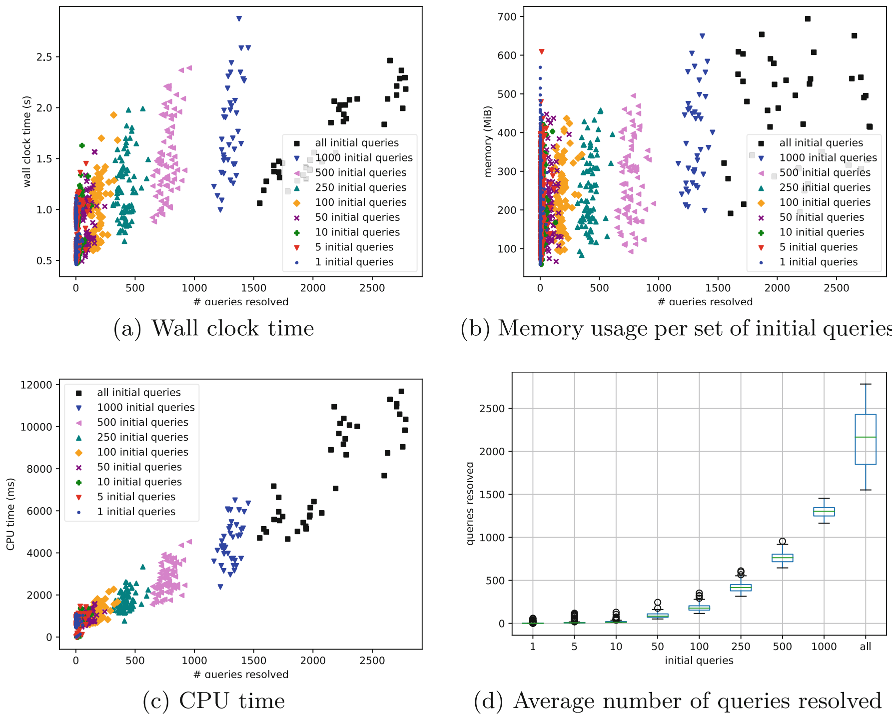
```
1 function fun57(arg57) {
2     ((chainTarget106)((fun57)((retAFunc100)(someIdentifier))))(someIdentifier
          );
3 }
4 function retAFunc100(arg615) {
5     return chainTarget106;
6 }
7 function chainTarget106(arg614) {}
```

Fig. 8. Example output by QuickCheck-based benchmark generator

Results. We ran the experiments on an AWS EC2 instance of type `c5.4xlarge` with Intel Xeon Platinum 8124M CPU with 16 cores and 32 GiB memory. We use Correto version 17.0.6.10.1 with a stack size limit of 512 MiB and heap size limit of 16 GiB. For each program, Merlin's analysis is run multiple times, with increasing the number of initial call graph queries in each iteration. The set of initial call graph queries is constructed by randomly selecting call sites occurring in the benchmark programs. This simulates a client analysis that issues queries for a subset of all call sites in the program. In the limit, issuing a query for every function call approximates a whole-program analysis. Our experiments also simulate a whole-program analysis by querying all call sites in the program.

The results of running Merlin on the synthetic benchmarks are shown in Fig. 9. Overall, the wall clock time (Fig. 9a) grows super-linearly with the number of resolved queries. The number of queries that need to be resolved (Fig. 9d) increases with the number of initial queries, matching the intuitive expectation that on-demand call graph construction explores only a part of the program on our benchmark set. Similarly, the wall clock time increases with the number of resolved queries, albeit to a lesser extent due to the use of parallelism. Memory consumption (Fig. 9b) remains relatively constant, indicating a significant fixed memory cost in our implementation.

As shown in Fig. 9a, whole-program analysis results (indicated by black boxes) often require less wall clock time to resolve than smaller initial sets of queries. This effect is due to the use of parallelism in the prototype: As demonstrated by Fig. 9c, whole-program analysis runs require as much or more CPU time to be resolved. However, but due to starting the analysis for all queries in parallel, they make better use of available CPU cores in the same span of wall clock time. This effect is somewhat in line with intuitive expectations: If a smaller set of queries is requested, there are less unrelated data flows to analyze, lowering the opportunities for parallelism. On the contrary, a whole-program analysis benefits from parallelism because many paths through a program can be analyzed independently. We double-check this explanation by reporting single-threaded results on the same benchmark set in Appendix A. This observation allows client analyses to fine-tune the strategy for call graph construction depending on the



(a) Wall clock time

(b) Memory usage per set of initial queries

(c) CPU time

(d) Average number of queries resolved

Fig. 9. Running MERLIN on synthetic benchmarks. The x-axis shows the size of each initial query set. The data points depict executions on different benchmark files. For each initial query set size, the same file is randomly sampled multiple times. Each initial query set is run independently without keeping intermediate results between runs.

scenario. On an end-user machine, using all available CPU cores may degrade the overall system performance too much to be viable, making on-demand analysis preferable. Electricity usage, environmental concerns, and battery life are other factors that make reducing CPU time relevant.

While the reduction in wall clock time based on the number queries to be resolved is often not significant compared to a whole-program analysis with the same technique, this data provides evidence that only a part of the program needs to be explored in order to answer a limited set of call graph queries. This effect may become more relevant in very large code bases or when using highly precise, expensive data flow analyses.

Threats to Validity. The memory usage reported in Fig. 9 is subject to measurement inaccuracies. CPU time and memory usage were measured using JVM internals with varying levels of guarantees. For example, memory usage is measured by first asking the JVM to perform garbage collection via `System.gc()`, but this is not guaranteed to garbage-collect all unreachable objects in the JVM

heap. As a result, memory usage may include state produced by previous analysis batches. Additionally, while all the internal state of all SPDS solvers is retained when measuring the memory consumption, there may be temporary data that is garbage-collected before the memory measurement is taken.

Limitations. As supporting the whole JavaScript langauge in SPDS is out of scope for this paper, MERLIN currently does not support all JavaScript language features, motivating evaluation on synthetic benchmarks that may not be representative of real-world JavaScript code. Instead, MERLIN presents an initial evaluation of whether this approach can be implemented using a realistic state-of-the-art data flow analysis. As a result, the above experiments do not show how much time is saved on real-world code, given the fact that many common JavaScript features are deliberately not used in the synthetic benchmark code, and the generated programs may use patterns that may not translate to patterns found in real-world code. Nevertheless, the subset of JavaScript that MERLIN supports and the set of synthetic benchmarks is sufficient to show the usefulness of on-demand call graph construction. We list the current limitations of MERLIN below, and consider addressing them as part of future work.

In the current implementation, MERLIN does not support dynamic property access, prototype inheritance, reflection, and JavaScript builtins. This may produce unsound results in practice. Moreover, context sharing between different queries is limited, even though this is in principle supported by the SPDS approach; this results in lower precision of our results than necessary. Since MERLIN reuses the TAJS [17] intermediate interpretation, it can only directly analyze EcmaScript 5 [11] programs, which in practice can be mitigated by transpiling code written in newer EcmaScript dialects using tools such as Babel [1].

Finally, MERLIN produces a large number of conceptually unnecessary queries, SPDS represents possible call stacks abstractly using a push-down system, where the system's stack contains program locations. To avoid querying for call sites when reaching a function boundary, the pushdown system could be consulted to approximate the possible elements at the top of the stack at this location. While SPDS constructs another automaton encoding the reachable configurations of the pushdown system using an existing approach [3,13], it is unclear whether this automaton allows extracting the required information. We leave leveraging call stack abstraction of SPDS to support this as future work.

5 Related Work

Demand Control-Flow Analysis [15] (Demand-CFA) tackles the problem of performing the functional equivalent of on-demand call graph construction for a purely functional lambda calculus, and similarly divides its approach into interdependent forward and backward queries. The key distinguishing feature of our work is providing a parameterized construction leveraging data flow analyses to support impure languages with non-functional features. In contrast, Demand-CFA fixes the specific analyses used for resolving expressions and finding call

sites. This approach is sufficient in the context of a purely functional language, but translation to a language with imperative features introduces a large design space of how to trade-off precision and scalability. By providing a parameterized approach, we sidestep such trade-offs and provide a modular building block.

Another line of work aims to make whole-program call graph construction scalable enough to apply to large code bases. A prominent example is Class-Hierarchy Analysis [7] (CHA) and subsequent work [2] in the context of object-oriented languages. CHA achieves scalability by making use of nominal typing to resolve higher-order behavior resulting from dynamic dispatch. Since all subtyping relationships are explicit in the syntax (for example, in the form of `extends` and `implements` clauses in Java), this is straightforward to compute efficiently. However, this approach is harder to apply to languages that use functions as first-class values without potentially introducing a large amount of imprecision. For example, given a higher-order function accepting a function of type `int -> int` as input, considering each function with this type (out of potentially many) as a potential callee in the body of the higher-order function might lead to many spurious call edges in practice. Using functions as first-class values is common practice in JavaScript, and is becoming common in more languages, for example through Java's introduction of lambda expressions [25] and streams [24]. Similarly, fast and scalable approaches exist for whole-program JavaScript call graph construction. Feldthaus et al. [14] present an underapproximate call graph construction algorithm for JavaScript that scales to usage inside an integrated development environment (IDE), which places strict requirements on how fast the analysis can be performed. However, in order to achieve this level of performance, the approach is intentionally unsound and misses call edges. Nielsen et. al [22] also present a highly scalable approach to call graph construction sacrificing soundness in some cases. While our implementation is also unsound in the presence of the same features that cause unsoundness in their work, our theoretical approach provides strong soundness guarantees.

Another well-known approach is variable-type analysis (VTA) [33], which produces reasonably scalable whole-program call graphs in the presence of higher-order functions and heap objects without requiring deep interleavings between the call graph construction and the data flow analysis. However, VTA's performance may render it too slow in certain contexts, e.g. for in-IDE use on large applications. To achieve this level of scalability, VTA's precision is constrained by its heap abstraction, while our approach allows for the use of more precise heap abstractions while hopefully remaining scalable for large code bases.

A key motivation for our work is the common theme of other analysis approaches assuming a precomputed call graph. Such examples include practical bug detection tools such as Infer [4], as well as theoretical results on Demanded Abstract Interpretation [32] that allow turning whole-program analyses into demand-driven analyses transparently. The latter example in particular may allow turning a whole-program data flow analysis into a fully demand-driven analysis by (i) obtaining an on-demand, but still call-graph-dependent, data flow

analysis by applying Demanded Abstract Interpretation, and (ii) lifting the call graph requirement using the approach presented in this paper.

Our work relies on the existence of sufficiently precise on-demand data flow analyses, an area that has seen improvements recently. Notably, Synchronized Push-Down Systems [30] reconcile the conflict between precise tracking of field accesses and calling contexts. Boomerang [31] provides another on-demand data flow analysis supporting exactly the same forward and backward queries required to instantiate our approach.

Our approach of issuing additional queries that allow each other to make progress in a mutually recursive fashion is inspired by Datalog-based analyses such as Doop [29] and CodeQL [16]. Datalog analyses, however, directly build a call graph together with a specific points-to analysis and do not typically allow plugging in another points-to analysis instead. Further, we are not aware of on-demand analyses implemented in Datalog. The formalization of our approach may also provide a starting point to reason about soundness of Datalog-based analyses, which has not been extensively studied formally.

6 Conclusions

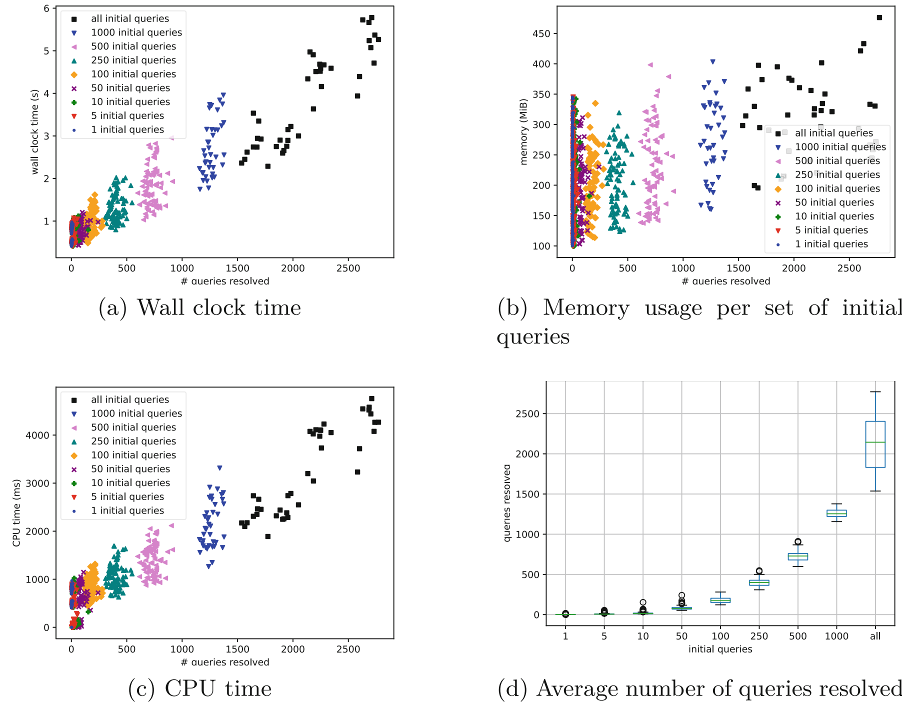
We present an approach for bootstrapping an on-demand call graph, leveraging underlying forward and backward data flow analyses. Our approach is parametric in the underlying analyses assuming only a notion of soundness up to a partial call graph. Based on this notion of soundness, we formalize our call graph construction and prove it sound (mechanized in Isabelle/HOL). Our prototype Merlin implements this approach for a subset of JavaScript using Synchronized Push-Down Systems [30] for both forward and backward data flow analysis. We evaluate Merlin on a synthetic benchmark set. The results indicate that on-demand call graph construction indeed has the potential to improve scalability by only exploring the relevant part of programs in the benchmark.

Acknowledgments. This paper describes work performed in part while David Seekatz was an Applied Scientist Intern at Amazon. Franco Raimondi holds concurrent appointments at Middlesex University and as an Amazon Scholar. Bor-Yuh Evan Chang holds concurrent appointments at the University of Colorado Boulder and as an Amazon Scholar. This paper describes work performed at Amazon and is not associated with Middlesex University nor the University of Colorado Boulder.

We are particularly grateful to Fangyi Zhou and Martin Schaef for their discussions and feedback on several drafts of this paper. We thank the anonymous reviewers for their helpful comments and feedback. This research was conducted in the Prime Video Automated Reasoning team and we are grateful to the entire team for their support.

A Single-Threaded Performance Results

Figure 10 shows the benchmark results when run on a single core, demonstrating that the faster whole-program results are caused by better CPU utilization.



(a) Wall clock time

(b) Memory usage per set of initial queries

(c) CPU time

(d) Average number of queries resolved

Fig. 10. Single-threaded performance results

References

1. Babel: Babel. https://babeljs.io/. Accessed 01 Apr 2023
2. Bacon, D.F., Sweeney, P.F.: Fast static analysis of C++ virtual function calls. In: Anderson, L., Coplien, J. (eds.) Proceedings of the 1996 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA 1996), San Jose, California, USA, 6–10 October 1996, pp. 324–341. ACM (1996)
3. Bouajjani, A., Esparza, J., Maler, O.: Reachability analysis of pushdown automata: application to model-checking. In: Mazurkiewicz, A., Winkowski, J. (eds.) CONCUR 1997. LNCS, vol. 1243, pp. 135–150. Springer, Heidelberg (1997). https://doi.org/10.1007/3-540-63141-0_10
4. Calcagno, C., Distefano, D.: Infer: an automatic program verifier for memory safety of C programs. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 459–465. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20398-5_33
5. Claessen, K., Hughes, J.: Quickcheck: a lightweight tool for random testing of Haskell programs. In: Odersky, M., Wadler, P. (eds.) Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP 2000), Montreal, Canada, 18–21 September 2000, pp. 268–279. ACM (2000)

6. CodeShield: de.fraunhofer.iem.SPDS. https://github.com/codeshield-security/spds. Accessed 30 Jan 2022
7. Dean, J., Grove, D., Chambers, C.: Optimization of object-oriented programs using static class hierarchy analysis. In: Tokoro, M., Pareschi, R. (eds.) ECOOP 1995. LNCS, vol. 952, pp. 77–101. Springer, Heidelberg (1995). https://doi.org/10.1007/3-540-49538-X_5
8. Dillig, I., Dillig, T., Aiken, A.: Sound, complete and scalable path-sensitive analysis. In: Gupta, R., Amarasinghe, S.P. (eds.) Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, 7–13 June 2008, pp. 270–280. ACM (2008)
9. Distefano, D., Fähndrich, M., Logozzo, F., O'Hearn, P.W.: Scaling static analyses at Facebook. Commun. ACM **62**(8), 62–70 (2019). https://doi.org/10.1145/3338112
10. Diwan, A., McKinley, K.S., Moss, J.E.B.: Type-based alias analysis. In: Davidson, J.W., Cooper, K.D., Berman, A.M. (eds.) Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada, 17–19 June 1998, pp. 106–117. ACM (1998)
11. ECMA International: ECMAScript language specification, 5th edition (2011). https://www.ecma-international.org/ecma-262/5.1/
12. Elliott, C., Hudak, P.: Functional reactive animation. In: Jones, S.L.P., Tofte, M., Berman, A.M. (eds.) Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP 1997), Amsterdam, The Netherlands, 9–11 June 1997, pp. 263–273. ACM (1997)
13. Esparza, J., Hansel, D., Rossmanith, P., Schwoon, S.: Efficient algorithms for model checking pushdown systems. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 232–247. Springer, Heidelberg (2000). https://doi.org/10.1007/10722167_20
14. Feldthaus, A., Schäfer, M., Sridharan, M., Dolby, J., Tip, F.: Efficient construction of approximate call graphs for JavaScript IDE services. In: Notkin, D., Cheng, B.H.C., Pohl, K. (eds.) 35th International Conference on Software Engineering, ICSE 2013, San Francisco, CA, USA, 18–26 May 2013, pp. 752–761. IEEE Computer Society (2013)
15. Germane, K., McCarthy, J., Adams, M.D., Might, M.: Demand control-flow analysis. In: Enea, C., Piskac, R. (eds.) VMCAI 2019. LNCS, vol. 11388, pp. 226–246. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-11245-5_11
16. GitHub: CodeQL. https://codeql.github.com/. Accessed 29 Jan 2022
17. 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). https://doi.org/10.1007/978-3-642-03237-0_17
18. Landman, D., Serebrenik, A., Vinju, J.J.: Challenges for static analysis of Java reflection: literature review and empirical study. In: Uchitel, S., Orso, A., Robillard, M.P. (eds.) Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, 20–28 May 2017, pp. 507–518. IEEE/ACM (2017)
19. Lhoták, O., Hendren, L.: Context-sensitive points-to analysis: is it worth it? In: Mycroft, A., Zeller, A. (eds.) CC 2006. LNCS, vol. 3923, pp. 47–64. Springer, Heidelberg (2006). https://doi.org/10.1007/11688839_5
20. Li, Y., Tan, T., Xue, J.: Understanding and analyzing Java reflection. ACM Trans. Softw. Eng. Methodol. **28**(2), 7:1-7:50 (2019)
21. Meta: React. https://reactjs.org/. Accessed 06 Feb 2022

22. Nielsen, B.B., Torp, M.T., Møller, A.: Modular call graph construction for security scanning of Node.js applications. In: Cadar, C., Zhang, X. (eds.) 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2021, Virtual Event, Denmark, 11–17 July 2021, pp. 29–41. ACM (2021)
23. Nipkow, T., Wenzel, M., Paulson, L.C.: 5. the rules of the game. In: Nipkow, T., Wenzel, M., Paulson, L.C. (eds.) Isabelle/HOL. LNCS, vol. 2283, pp. 67–104. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45949-9_5
24. Oracle: Java streams. https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html. Accessed 02 Feb 2022
25. Oracle: Lambda expressions for the Java programming language (2014). https://jcp.org/aboutJava/communityprocess/final/jsr335/index.html
26. Sadowski, C., Aftandilian, E., Eagle, A., Miller-Cushon, L., Jaspan, C.: Lessons from building static analysis tools at google. Commun. ACM **61**(4), 58–66 (2018). https://doi.org/10.1145/3188720
27. Schoepe, D.: Lifting on-demand analysis to higher-order languages (artifact). Static Anal. Symp. (2023). https://doi.org/10.5281/zenodo.8189312
28. Smaragdakis, Y., Balatsouras, G., Kastrinis, G., Bravenboer, M.: More sound static handling of Java reflection. In: Feng, X., Park, S. (eds.) APLAS 2015. LNCS, vol. 9458, pp. 485–503. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-26529-2_26
29. Smaragdakis, Y., Bravenboer, M.: Using datalog for fast and easy program analysis. In: de Moor, O., Gottlob, G., Furche, T., Sellers, A. (eds.) Datalog 2.0 2010. LNCS, vol. 6702, pp. 245–251. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24206-9_14
30. Späth, J., Ali, K., Bodden, E.: Context-, flow-, and field-sensitive data-flow analysis using synchronized pushdown systems. Proc. ACM Program. Lang. **3**(POPL), 48:1–48:29 (2019)
31. Späth, J., Do, L.N.Q., Ali, K., Bodden, E.: Boomerang: demand-driven flow- and context-sensitive pointer analysis for Java. In: Krishnamurthi, S., Lerner, B.S. (eds.) 30th European Conference on Object-Oriented Programming, ECOOP 2016, 18–22 July 2016, Rome, Italy. LIPIcs, vol. 56, pp. 22:1–22:26. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2016)
32. Stein, B., Chang, B.E., Sridharan, M.: Demanded abstract interpretation. In: Freund, S.N., Yahav, E. (eds.) 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2021, Virtual Event, Canada, 20–25 June 2021, pp. 282–295. ACM (2021)
33. Sundaresan, V., et al.: Practical virtual method call resolution for Java. In: Rosson, M.B., Lea, D. (eds.) Proceedings of the 2000 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications, OOPSLA 2000, Minneapolis, Minnesota, USA, 15–19 October 2000, pp. 264–280. ACM (2000)