Interactive Abstract Interpretation with Demanded Summarization

BENNO STEIN*, SkipLabs, UK & University of Colorado Boulder, USA
BOR-YUH EVAN CHANG†, University of Colorado Boulder & Amazon, USA
MANU SRIDHARAN, University of California, Riverside, USA

We consider the problem of making expressive, interactive static analyzers *compositional*. Such a technique could help bring the power of server-based static analyses to integrated development environments (IDEs), updating their results live as the code is modified. Compositionality is key for this scenario, as it enables reuse of already-computed analysis results for unmodified code. Previous techniques for interactive static analysis either lack compositionality, cannot express arbitrary abstract domains, or are not from-scratch consistent.

We present demanded summarization, the first algorithm for incremental compositional analysis in arbitrary abstract domains which guarantees from-scratch consistency. Our approach analyzes individual procedures using a recent technique for demanded analysis, computing summaries on demand for procedure calls. A dynamically-updated summary dependency graph enables precise result invalidation after program edits, and the algorithm is carefully designed to guarantee from-scratch-consistent results after edits, even in the presence of recursion and in arbitrary abstract domains. We formalize our technique and prove soundness, termination, and from-scratch consistency. An experimental evaluation of a prototype implementation on synthetic and real-world program edits provides evidence for the feasibility of this theoretical framework, showing potential for major performance benefits over non-demanded compositional analyses.

CCS Concepts: • **Theory of computation** → **Program analysis**; • **Software and its engineering** → **Formal software verification**.

Additional Key Words and Phrases: Abstract interpretation, Incremental computation

1 INTRODUCTION

This paper addresses the problem of developing incremental and demand-driven static analyzers that simultaneously support real-time user interaction, arbitrarily complex abstract domains, and compositional analysis. Static analysis is being increasingly deployed for bug finding and verification in continuous integration (CI) pipelines and automated code review systems [Distefano et al. 2019; Sadowski et al. 2018]. However, the cost of analyzing large programs means that developers must wait minutes or sometimes hours to receive analysis results after making changes, depressing fix rates and ultimately limiting the effectiveness of these tools.

In this paper, we describe an analysis engine that is (1) *incremental* in that it reuses analysis results unaffected by program edits, (2) *demand-driven* in that it performs only the analysis work needed to respond to client-issued queries, and (3) *compositional* in that it produces reusable and composable procedure summaries for interprocedural analysis. Furthermore, our framework makes no demands on analysis implementors beyond those typical of a batch abstract interpretation engine — specifically, it supports arbitrary abstract domains, including infinite-height domains with non-monotonic widening operators.

We formalize incrementality and demand for summary-based abstract interpretation and show that our approach is sound, terminating, and *from-scratch consistent*, i.e., that analysis results are

*This paper describes work performed at CU Boulder and is not associated with SkipLabs.
†Bor-Yuh Evan Chang holds concurrent appointments at the University of Colorado Boulder and as an Amazon Scholar. This paper describes work performed at CU Boulder and is not associated with Amazon.

Authors' addresses: Benno Stein, SkipLabs, UK & University of Colorado Boulder, USA, benno@skiplabs.io; Bor-Yuh Evan Chang, University of Colorado Boulder & Amazon, USA, evan.chang@colorado.edu; Manu Sridharan, University of California, Riverside, USA, manu@cs.ucr.edu.

always at least as precise as reanalyzing the program from scratch. Soundness and termination are familiar metatheoretic properties from the program analysis literature, whereas from-scratch consistency has been called "the fundamental correctness property of incremental computation" in general incremental computation literature [Hammer et al. 2015]. In the context of interactive program analysis, from-scratch consistency guarantees that there is *no loss of precision or flakiness* introduced by incremental or demand-driven infrastructure.

Recent work offers an approach for *interactive analysis*, making analyses with arbitrary abstract domains incremental and demand-driven by explicitly reifying abstract interpretation computation into acyclic dependency structures called *demanded*[1] abstract interpretation graphs (DAIGs) [Stein et al. 2021a]. This combination of incrementality and demand is desirable for scenarios requiring interactive performance: it avoids unnecessary recomputation of unchanged outputs when inputs change, and also avoids unnecessary computation of outputs that are never needed by the client [Hammer et al. 2015, 2014].

Although this technique computes analysis results efficiently and interactively, it is limited to *intraprocedural* analysis of imperative programs. Stein et al. [2021a] describe an informal extension of their whole-program operational approach to interprocedural analysis, but this extension does not handle recursion at all and is known to have issues in scaling up to large programs. Instead, a standard approach to scale batch abstract interpretation to large programs is to compute procedure summaries and then apply these summaries to analyze the whole program compositionally [Blackshear et al. 2018; Calcagno and Distefano 2011; Distefano et al. 2019; Reps et al. 1995; Schubert et al. 2021].

Ideally, an interactive demanded abstract interpreter could interoperate with a batch compositional analysis seamlessly, updating analysis results locally while drawing on existing procedure summaries computed on CI servers—with the same arbitrarily complex abstract domains.

Traditional compositional analyses based on the well-known *tabulation* algorithm combine "operational" dataflow analysis (i.e., how to *interpret* commands intraprocedurally to compute the analysis state at a program point, given the analysis states at predecessor points) with "denotational" procedure summaries mapping code to a functional or relational abstraction of its semantics (i.e., a *compilation* of code to a transformer on analysis states for interprocedural analysis) [Reps et al. 1995; Sharir and Pnueli 1981], but existing dependency-based approaches to incremental or demand-driven analysis tend to focus on one or the other style of analysis.

In this paper, we show how demanded analysis infrastructure can be extended to tabulation-based compositional analysis in the presence of arbitrary abstract domains and recursive procedures, while providing meta-theoretical guarantees including soundness, termination, and *from-scratch consistency* of analysis results. From-scratch consistency ensures that incremental results agree precisely with an underlying batch analysis, and is crucial for reliable deployment in analysis systems that strive for deterministic and reproducible results.

To address this challenge, we introduce a dependency map that is dynamically extended to capture the dependencies that arise from using demanded procedure summaries. These dynamic dependencies also enable our algorithm to detect the self-referential procedure summaries that arise when analyzing recursive procedures and require a further fixed-point iteration.

Our framework improves over the state-of-the-art in multiple ways. Compared to previous work on intraprocedural demanded analysis [Stein et al. 2021a], which rely on the restricted structure of loops in reducible control flow graphs, our approach naturally handles the richly structured dependency graphs that arise from analyzing interprocedural control flow with recursion.

[1] We use "demanded" to describe a computation that is *both* incremental and demand-driven, following Hammer et al. [2015, 2014] and Stein et al. [2021a], all of which reify dependency graphs to enable real-time interactivity.

Supporting the analysis of recursive procedures is particularly important in languages with higher-order or dynamic dispatch, as it is well-known that reasonable approximation in the underlying call graph construction can lead to recursion in the program abstraction.

Other recent compositional analysis frameworks [Blackshear et al. 2018; Calcagno and Distefano 2011] have claimed some degree of incrementality, so as to reduce analysis times on CI servers after a code change. It is true that compositionality naturally yields some degree of incrementality, but tracking the invalidation of summaries is tricky and can lead to subtle soundness bugs, especially surrounding virtual calls and incremental changes to dependency structures. Such issues have manifested in the Infer static analyzer, for example, due to insufficiently conservative dependency tracking or invalidation of pre-edit summaries using the post-edit dependency structure [Stein 2023]. These works give no formal treatment of incrementality, and are typically deployed in a non-incremental configuration. A variant of the technique described in this paper has recently been implemented in the Infer static analyzer, enabling deployment of incremental analysis and yielding significant (on the order of 3x) analysis speedups in CI [Stein 2023].

This paper aims to formalize and provide a richer understanding of the dependency management problem that underlies sound and precise incremental and demand-driven analysis.

To our best knowledge, this paper presents the first algorithm for incremental compositional analysis in arbitrary abstract domains (including infinite-height domains with non-monotonic widening). Further, it supports demand-driven queries for analysis results, and we provide a full meta-theory showing soundness, termination, and from-scratch consistency in the presence of recursive procedures.

In short, we make the following key contributions:

- We introduce a framework for interactive abstract interpretation with *demanded summarization*, reifying the dependency structure of a tabulation-based analysis as a dynamically-evolving *demanded summarization graph* (DSG) (Section 4). The result is a demand-driven interprocedural analysis framework that is both incremental and compositional by default.
- We formalize such an analysis as an on-demand evaluation of DSGs, showing that it is sound, terminating, and from-scratch consistent with the underlying batch analysis (Section 5).
- We describe a prototype implementation of the framework and evaluate it on both synthetic benchmarks and real bug-fixing edits from open-source Java programs. Our proof-of-concept evaluation finds that our framework enables real-time interprocedural analysis in rich abstract domains (Section 6).
- We formalize extensions to the framework that enable memory and computation saving optimizations without compromising its formal guarantees (Section 7). While seemingly simple, we see that maintaining the summary dependency structure in a sound way is surprisingly subtle.

2 OVERVIEW

In this section, we illustrate demanded summarization by example. Fig. 1 shows a numerical program (adapted from Sagiv et al. [1996]) in an imperative language, with a recursive procedure `p`. Fig. 1 also shows a simple example *edit* on line 3 of the program, designed to demonstrate some of our technique's key features.

Note that throughout this worked example, we assume a standard semantics for variable scoping and formal/actual parameter binding; our formalism elides these details for the sake of simplicity and clarity as they are unrelated to the developments in this paper.

For illustration, we consider analysis of the original and edited programs using an *interval* abstract domain to bound variable ranges – a textbook example of an infinite height domain with a non-monotonic widening operator [Cousot and Cousot 1977]. With this domain, the analysis

```
1  int x = 0;
2  void main() {
3  - int n = 3;
   + int n = 5;
4    p(n);
5    print(x);
6  }
```

```
7  void p(int a) {
8    if (a > 0) {
9      a -= 2;
10     p(a);
11     a += 2;
12   }
13   x = -2 * a + 5;
14 }
```

Fig. 1. An example program written in an imperative language with recursive procedures, adapted from Sagiv et al. [1996], along with an edit applied at line 3.

can prove, that $x = -1$ at the exit of `main` originally, and $x = -5$ at exit after the edit. An efficient incremental analysis should re-use many intermediate results from analysis of the original program when re-analyzing the edited version.

Our key goal is to define an analysis framework that simultaneously supports incremental, demand-driven, and compositional analysis in arbitrary abstract domains. Recent work by Stein et al. [2021a] supports incremental and demand-driven analysis in arbitrary domains, but only for *intraprocedural* analysis; they present an informal extension for interprocedural analysis but it is not compositional and cannot handle recursion.

We first show the drawbacks of a lack of compositionality for our example, then show how our *demanded summarization* approach achieves both compositionality and support for recursion. This technique enables the use of Stein et al.'s [2021a] fine-grained demanded abstract interpreters at scale, so we describe the approach using DAIGs to be concrete in our presentation and support incrementality at the fine granularity of statements. However, the same general approach could be used to derive an analysis with a coarser procedure-level granularity if instantiated with a standard batch intraprocedural abstract interpreter.

2.1 Incrementality and Context Sensitivity

Fig. 2 illustrates how an *operational*[2] approach to interprocedural analysis has disadvantages for incremental analysis. In the operational call-strings approach [Sharir and Pnueli 1981], a procedure is analyzed separately for each abstract call stack, or call string, in which it is called. Fig. 2 shows the analysis result for the original program using call strings of length 1; `p` is analyzed once for each call site (lines 4 and 10). Using this context-sensitivity policy provides sufficient precision to prove $x = -1$ at exit.

The downside of the operational approach comes in doing incremental analysis. The operational approach indexes its results on the calling context under which a procedure is invoked. Hence, if an edit leads to recomputing facts at some call site of a procedure, the procedure must be re-analyzed for corresponding contexts. In Fig. 1, the edit on line 3 causes re-computation of the abstract facts before the call site at line 4. This in turn necessitates invalidation of all analysis facts computed in `p` for this call (shown with ✗). Similarly, all facts for `p` from the recursive call must be invalidated. This invalidation is wasteful: `p` was not modified and it invokes no other functions (besides itself), so results for `p` should be unaffected by the change.

[2] We use "operational" and "denotational" (following Jeannet et al. [2010]), which are analogous to but more general than the commonly-used "call-strings" and "functional" terminology of Sharir and Pnueli [1981]. In particular, (1) call-strings are just one instance of a broader class of context-sensitive interprocedural analyses that operationally propagate predicates along a control-flow graph until a fixed-point is reached, and (2) summaries need not be functions but can be any abstraction of a procedure's denotational semantics.

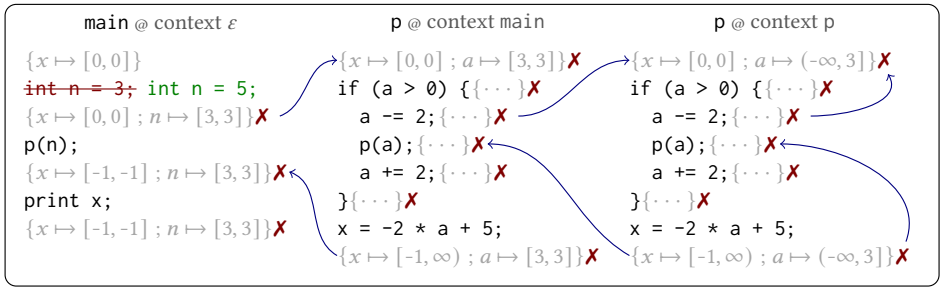


Fig. 2. The results of a 1-call-string sensitive abstract interpretation (typifying the *operational* approach to interprocedural analysis) of the program in an interval domain [Cousot and Cousot 1977]. Those abstract states invalidated by the line-3 edit are marked by a ✗, and blue arrows represent interprocedural analysis dependencies.

A *denotational*[2] interprocedural analysis computes a transformer for each procedure – typically, some two-state relational abstraction or Hoare triple(s) over the procedure, which can then be applied at callsites when the abstract state is compatible with a summary's initial state. This is the "functional approach" of Sharir and Pnueli [Sharir and Pnueli 1981], and such analyses are often referred to as "summary-based," "compositional," or "modular." Note that a denotational interprocedural analysis need not analyze each procedure precisely once, but may compute and tabulate multiple partial procedure summaries.

Fig. 3 shows the result of our demanded summarization approach, which is a denotational, tabulation-based approach to interprocedural analysis for our example. For the moment, ignore the edit and arrows between the boxes, which are part of explaining our approach further below. What to note now is that copies of `p` are distinguished by the abstract fact reaching `p`'s entry, e.g., $\{x \mapsto [0,0]; a \mapsto [3,3]\}$. In this summary-based approach, clearly editing line 3 does not impact the results for `p`, as the tabulated summaries are independent of specific callers. Hence, only results in `main` need to be invalidated. Our challenge lies in adapting the denotational approach to build incremental and demand-driven analyses for arbitrary abstract domains with interactive performance and provable correctness and precision guarantees.

2.2 Demanded Abstract Interpretation

In the work of Stein et al. [2021a], incremental and demand-driven analysis is achieved via demanded abstract interpretation graphs, or DAIGs. These structures reify analysis computation in a graph that makes dependencies among analysis results and program statements explicit, and support two key analysis operations: *queries* and *edits*. A *query* can be raised for the analysis result at some program point p. The query is answered by computing analysis results at all points backward-reachable from p in the DAIG, which captures all dependencies. Intermediate results are cached to speed up computation of future queries. In response to a program edit, analysis results that are forward-reachable from the edit point(s) in the DAIG (those results dependent on edited statements) are *dirtied*; these dirtied results may be recomputed in response to a future query.

These techniques – and the "demanded" terminology[1] – draw heavily on the incremental computation literature and can be seen as specializing traditional graph-based incremental computation techniques to the setting of intraprocedural abstract interpretation [Hammer et al. 2015, 2014; Stein et al. 2021a].

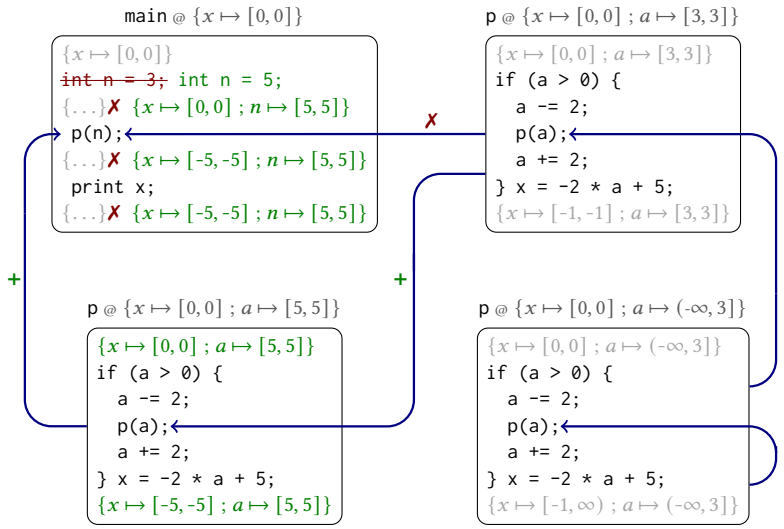


Fig. 3. Demanded summarization applied to the program and edit of Fig. 1, with some non-procedure-entry/exit abstract states in the summaries of p elided for clarity. As in Fig. 2, blue edges denote interprocedural analysis dependencies.
After the edit at line 3, the previously-computed summaries of p (on the right half of this figure) are still valid and can be used to produce the additional p summary needed to reanalyze `main`.
The summary dependency edge labeled by a ✗ is removed when the callsite it points to is dirtied by the edit. When a new query is issued at the exit of `main`, only the green states must be recomputed, instantiating a new partial summary of p and two new dependencies (labeled by +) in the process.

We will avoid delving into the details and complexities of intraprocedural analysis in this paper for clarity of presentation and to show that demanded summarization has no fundamental dependency on the specific intraprocedural analysis used to produce summaries.

2.3 Demanded Summarization

Our new approach to incremental, demand-driven, compositional interprocedural analysis is based on *demanded summarization graphs* (DSGs), which embody three key ideas. First, each denotational procedure summary is computed using a DAIG, which is a node in the DSG. Each individual DAIG is guaranteed to be from-scratch consistent [Stein et al. 2021a]. Second, at a procedure call, a new DAIG summary must be computed on demand; an edge in the DSG tracks where the summary is applied. To ensure from-scratch consistency, a procedure summary is only applied at call sites *after* computation of that summary has converged. Finally, recursion is handled via an additional fixed-point computation *within the recursive procedure*, designed carefully to maintain from-scratch consistency.

The key difficulty addressed by the DSG semantics is the interleaving of intraprocedural analysis (within DAIGs) and interprocedural analysis (orchestrated amongst a collection of DAIGs) in a way that is sound, terminating, and from-scratch consistent.

The dependency structures induced by general interprocedural control flow — wherein procedures may have many recursive calls in arbitrary position, possibly nested in loops and intermingled with calls to other procedures — require fundamentally different handling than cyclic intraprocedural control flow.

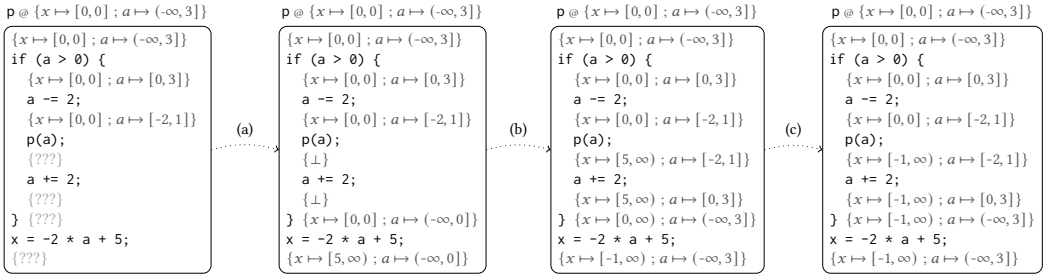


Fig. 4. In-place fixed-point computation of the self-referential summary of Fig. 1, showing how analysis converges on the control-flow cycle between its recursive return site and procedure exit location.

Fig. 3 shows how the example of Fig. 1 is analyzed using a DSG. Each box in the diagram represents a DAIG, while the blue edges represent interprocedural summary dependencies: a DSG essentially consists of a collection of independent procedure summary DAIGs overlaid by an interprocedural dependency graph.

Assume an initial query for the analysis state at the exit of `main` when $x = 0$ at `main`'s entry. Our approach first creates a corresponding DAIG (labeled `main` @ $\{x \mapsto [0, 0]\}$) to perform this analysis. Intraprocedural analysis proceeds until it reaches the call `p(n)` with fact $\{x \mapsto [0, 0]; n \mapsto [3, 3]\}$. Handling the call and finishing the analysis of `main` requires first computing a summary for `p`, so we instantiate a new DAIG for `p` (at top right of Fig. 3, after performing formal/actual parameter binding on the abstract state).

Recursion. Analysis within the DAIG for `p` with initial fact $\{x \mapsto [0, 0]; a \mapsto [3, 3]\}$ reaches a recursive call `p(a)`. Here, we recognize the call as recursive and apply widening before instantiating another DAIG over `p`, yielding initial state $\{x \mapsto [0, 0]; a \mapsto (-\infty, 3]\}$ (at bottom right of Fig. 3). Analysis in this new DAIG once again reaches the recursive callsite, but widening of the entry state has converged to $\{x \mapsto [0, 0]; a \mapsto (-\infty, 3]\}$.

Now, we must analyze the control-flow path(s) *after* the recursive call in the final (self-referential) DAIG, as we need a converged summary before we can propagate back to callers. Further, a fixed point may be required to converge on the exit state.

To obtain a converged exit state, we run a carefully-designed fixed-point analysis *within* the final DAIG, using intraprocedural dirtying to iterate. The process is illustrated step-by-step in Fig. 4.

It starts by injecting $\bot$ as the post-state of the final recursive call (step (a)), since no dataflow has yet reached the exit. Propagating to the exit yields a fact $\{x \mapsto [5, \infty]; a \mapsto (-\infty, 0]\}$, due to the join with the non-recursive path. To iterate, we dirty the post-state of the recursive call within the DAIG, update the post-state to this new fact, and then re-query the exit state (step (b)). We continue this process one more time (step (c)), and see the exit state converges to $\{x \mapsto [-1, \infty]; a \mapsto (-\infty, 3]\}$.

Finally, the fully-analyzed rightmost DAIG of Fig. 4 can be applied as a summary in the DAIG that initially demanded it (at top right of Fig. 3), which can then be fully analyzed and applied as a summary in the `main` DAIG. We add interprocedural dependency edges to the DSG to track each of these summary applications: the blue edges not marked by a + in Fig. 3. Finally, we finish analyzing `main` and return the query result $\{x \mapsto [-1, -1]; n \mapsto [3, 3]\}$.

Incremental Updates. Now, when the edit is made to `main`, its downstream dependencies (the states marked by ✗ in Fig. 3) are dirtied in the `main` DAIG, but no other DAIG is affected because the `main` summary has no dependencies. Suppose another query is issued for the exit abstract state of `main`: we now reach the `p(n)` callsite with no applicable summary and so instantiate a new DAIG for `p` with initial state $\{x \mapsto [0, 0]; a \mapsto [5, 5]\}$, at bottom left of Fig. 3. However, the

previously-computed summary[3] with precondition $\{x \mapsto [0,0]; a \mapsto [3,3]\}$ can now be used at the recursive callsite `p(a)`, allowing analysis to complete without recomputing a fixed point over the recursive procedure. Once again, summary applications are tracked by dependency edges to enable precise dirtying in response to future edits.

From-scratch consistency. From-scratch consistency requires computing *exactly* the same result on an edited program as a batch analysis. Hence, our approach strictly orders computation of analysis results, including integration of summaries into callers, enabling from-scratch consistency for arbitrary domains. The DSG algorithm guarantees that the requested summary for `p` will only be integrated into `main` once summary computation has completely converged.

In tabulation analyses like the IFDS algorithm [Reps et al. 1995], intermediate summary results may be propagated to callers before all paths through a procedure are completely analyzed. So, for `p`, a summary result for the non-recursive path could be propagated to callers before the recursive call is fully analyzed. For an IFDS problem, the variance in propagation ordering does not impact the final analysis result, since the join operation is always set union. However, since our framework supports infinite abstract domains with non-monotonic widening operators (like the interval domain), variance in propagation order could cause differences in the final analysis result.

Demanded Summarization and Compositional Analysis. When a new summary is required to analyze a procedure call, demanded summarization instantiates a new DAIG with the requisite initial abstract state and issues a query for the abstract state at its exit. Within an instantiated DAIG, we can reuse results to respond to future queries and efficiently dirty results in response to program edits. However, instantiated DAIGs cost memory, so a balance must be struck between the granularity of cached analysis results and memory usage.

We can tune this balance by condensing DAIGs to two-state summaries, remembering the entry and exit states — essentially a Hoare triple over the procedure — while discarding all intermediate analysis facts that contributed to said triple. This is formalized in Section 7, where we show that from-scratch consistency is preserved under this transformation.

A demanded summarization-based analysis can also be initialized with procedure summaries computed from a batch compositional analysis, provided that dependencies between summaries are preserved to enable dirtying after edits. This addresses the motivating problem of connecting server-based compositional analysis with interactive analysis in the IDE.

3 PRELIMINARIES: ABSTRACT INTERPRETATION

In this section, we fix syntax, semantics, and terminology for programs and abstract interpreters.

These are intentionally standard and define a typical interface for a batch summary-based batch abstract interpretation engine (see e.g. Padhye and Khedker [2013] for a similar formulation of the approach, which can be understood as an extension of Reps et al. [1995] to abstract interpretation). By design, all constructions are *generic* in the underlying abstract domain and statement language, and can be instantiated to a wide range of analysis problems and imperative programming languages. A procedure $\langle L, E\rangle$ is a reducible control-flow graph over an unspecified statement language *Stmt* with procedure call edges of the form $\ell \dashv[\texttt{call}\ \rho]\mapsto \ell'$, where ℓ is the location preceding the call and

[3]Note that this summary is actually more precise than the one that would have been computed by from-scratch batch analysis of the updated program. A DSG may apply a compatible summary whose precondition is stronger than the result of widening at recursive callsites without loss of precision, though its result may actually be more precise than from-scratch batch analysis if it does so. Although this optimization strictly improves analysis precision, it technically violates from-scratch consistency and as such is not applied in our formal system.

ℓ' is the return site:

$$\begin{array}{rcll} \text{statements} & s \in \mathit{Stmt} & & \\ \text{locations} & \ell \in \mathit{Loc} & & \\ \text{control-flow edges} & e \in \mathit{Edge} & ::= \ell \dashv [s] \mapsto \ell' & \\ & & \mid \quad \ell \dashv [\texttt{call}\ \rho] \mapsto \ell' & \\ \text{procedures} & \langle L, E \rangle \in \mathit{Proc} & = \mathcal{P}(\mathit{Loc}) \times \mathcal{P}(\mathit{Edge}) & \end{array}$$

A program is a labeled collection of such procedures with a distinguished "main" entry-point.

$$\begin{array}{rcl} \text{procedure labels} & \rho \in \mathit{Label} & \ni \rho_{\text{main}} \\ \text{programs} & P & : \mathit{Label} \hookrightarrow \mathit{Proc} \end{array}$$

We denote by L_P^ρ, E_P^ρ, $\text{entry}_P(\rho)$ and $\text{exit}_P(\rho)$ respectively the program-location set, control-flow edge set, entry location, and exit location of ρ. We also write ρ_P^ℓ for the unique ρ containing ℓ and E_P^* for the set of all control-flow graph (CFG) edges in P, and elide P subscripts where they are clear from context. This core language considers direct calls without parameters or return values for ease of presentation only and is not a fundamental limitation. In Section 6, we discuss implementation considerations when applying our approach to Java programs. However, note that our formal framework does not directly model mutual recursion or higher-order control flow.

Concrete Semantics. To define a concrete semantics of this language, we assume a denotational semantics for statements:

$$\begin{array}{rcl} \text{concrete states} & \sigma \in \Sigma & \text{(with initial state } \sigma_0\text{)} \\ \text{concrete semantics} & [\![\cdot]\!] & : \mathit{Stmt} \to \Sigma \to \Sigma_\bot \\ \text{concrete stacks} & \kappa \in K & ::= \varepsilon \mid e :: \kappa \end{array}$$

where the function $[\![\cdot]\!]$ gives a denotational interpretation of non-callsite program statements over concrete program states σ (with $\bot$ being an invalid state and $\Sigma_\bot = \Sigma \cup \{\bot\}$). Concrete call stacks κ are used to keep track of return sites in procedure calls. A state-collecting semantics $[\![\cdot]\!]_P^* : \mathit{Loc} \to \mathcal{P}(\Sigma \times K)$ for this language with procedure calls can be defined as a fixed-point over the concrete transfer function $[\![\cdot]\!]$.

Abstract Semantics. An abstract interpreter for *procedures* of this language is a tuple $\langle \Sigma^\sharp, \varphi_0, [\![\cdot]\!]^\sharp, \sqsubseteq, \sqcup, \nabla \rangle$ consisting of an abstract domain $\Sigma^\sharp$ equipped with distinguished initial state φ_0, partial order $\sqsubseteq$, join $\sqcup$, and widening ∇, as well as an abstract semantics (i.e., transfer function) $[\![\cdot]\!]^\sharp$ that interprets statements as functions over abstract states, all subject to the usual soundness conditions. That is, we require that (1) $\Sigma^\sharp$ forms a semi-lattice under $\sqsubseteq$ with join $\sqcup$ and bottom element $\bot$, (2) ∇ satisfies the ascending chain condition, and (3) φ_0 and $[\![\cdot]\!]^\sharp$ are sound with respect to their concrete counterparts σ_0 and $[\![\cdot]\!]$.

A solution to an intraprocedural abstract interpretation problem is a pre-fixed-point of the abstract transfer function over the flow graph $\langle L, E \rangle$, with the join $\sqcup$ applied at locations in L with in-degree ≥ 2 and the widening ∇ applied infinitely often on cycles. It is well-known that such a solution can be computed by worklist iteration, and that such a solution is sound if the transfer function is locally-sound [Cousot and Cousot 1977].

Just as we defined the concrete state-collecting semantics $[\![\cdot]\!]_P^*$ for this language with procedure calls as a fixed-point over the concrete transfer function, we can define an abstract state-collecting semantics $[\![\cdot]\!]_P^{\sharp *}$ as a fixed-point over abstract semantic functions. However, such a definition does not correspond to a computable analysis, since the space K of concrete stacks is unbounded. A standard solution to this issue is to apply the call-strings approach [Sharir and Pnueli 1981], abstracting concrete stacks by truncating instead of extending them indiscriminately in the equation for procedure entry locations.

3.1 Intraprocedural Demanded Abstract Interpretation

In this section, we define semantic judgments for intraprocedural demand-driven analysis and incremental edits, and make some assumptions about their behavior. This treatment is based on the demanded abstract interpretation graphs (DAIGs) of Stein et al. [2021a], but is intentionally abstract in the underlying details of intraprocedural analysis.

This "black-box" description is meant to facilitate a clear and understandable presentation of interprocedural analysis in the sections to follow, and to demonstrate that the demanded summarization algorithm does *not* depend on specific implementation details of DAIGs and generalizes in principle to other producers of procedure summaries.

A DAIG $\mathcal{D}$ reifies an abstract interpretation computation into a structure that supports interactive program analysis, that is, demand-driven queries and incremental edits [Stein et al. 2021a]. Partial analysis results are stored in reference cells that are assigned unique names $n ::= \underline{\ell} \mid n_1 \cdot n_2 \mid \ldots$. The location name $\underline{\ell}$ corresponds to the cell that, when demanded, stores the fixed-point invariant at location ℓ, and the product name $\underline{\ell} \cdot \underline{\ell'}$ stores the statement s (or procedure call $\texttt{call}\ \rho$) labeling the control-flow edge from ℓ to ℓ'. A DAIG is then a directed acyclic hypergraph where reference cells are nodes and where edges are labeled by abstract interpreter operations (e.g., $[\![\cdot]\!]^\sharp$, $\sqcup$, ∇) that specify the operation to apply when its output cell is queried. Demanded abstract interpretation is captured by two DAIG-rewriting judgments:

$$\begin{array}{rl} \text{demand-driven queries} & \mathcal{D} \vdash n \Rightarrow \varphi \,;\, \mathcal{D}' \\ \text{incremental edits} & \mathcal{D} \vdash n \Leftarrow v_\varepsilon \,;\, \mathcal{D}' \end{array}$$

Queries. The judgment form $\mathcal{D} \vdash n \Rightarrow \varphi \,;\, \mathcal{D}'$ is read as "a query for the abstract state named by n in DAIG $\mathcal{D}$ yields result φ by demanded abstract interpretation with updated DAIG $\mathcal{D}'$." A demanded abstract interpretation [Stein et al. 2021a] computes a result φ to store in the cell named by n by evaluating backwards-reachable dependencies of cell n in DAIG $\mathcal{D}$ while unrolling fixed-point computations on demand to maintain the DAIG acyclicity invariant and eventually resulting in $\mathcal{D}'$. We assume that this querying judgment is sound, terminating, and from-scratch consistent with respect to the underlying abstract interpretation.

Edits. The judgment form $\mathcal{D} \vdash n \Leftarrow v_\varepsilon \,;\, \mathcal{D}'$ is read as "an edit that writes value v (or the empty symbol ε) to the cell named by n in DAIG $\mathcal{D}$ yields updated DAIG $\mathcal{D}'$ with cells depending on n marked as 'dirty'." Demanded abstract interpretation supports incremental edits by eagerly "dirtying" those results forward-reachable from the edit to cell n in DAIG $\mathcal{D}$ (e.g., writing a statement s to a control flow edge from ℓ to ℓ' is given by $\mathcal{D} \vdash \underline{\ell} \cdot \underline{\ell'} \Leftarrow s \,;\, \mathcal{D}'$) . We assume that this dirtying judgment is conservative in the sense that all potentially-affected abstract states are dirtied. Since incremental dirtying is eager and demand-driven query evaluation is lazy, dirtied results are only recomputed if needed for a future query.

3.2 Summarization and Tabulation

Instead of abstracting concrete stacks and propagating abstract data flow in the resulting interprocedural control-flow graph, an alternative approach is to build compositional summaries for each procedure (i.e., the functional approach [Sharir and Pnueli 1981]). Procedure summaries can be built by defining relational, two-state abstract domains (e.g., [Jeannet et al. 2010; Yorsh et al. 2008]) or by tabulating pairs of abstract states [Padhye and Khedker 2013; Reps et al. 1995; Sharir and Pnueli 1981]. In either case, the goal is to compute procedure summaries $\{\varphi\}\ \rho\ \{\varphi'\}$ for each procedure ρ that can be applied at call sites $\ell \dashv [\texttt{call}\ \rho] \mapsto \ell'$.

A tabulation approach works by propagating intraprocedural dataflow through the procedure control-flow graph using the abstract transfer function $[\![\cdot]\!]^\sharp$, join $\sqcup$, and widen ∇ until it encounters

a procedure call, at which point either (1) corresponding procedure summaries have already been computed and can simply be applied, or (2) new summaries are required, so the process continues recursively. Convergence is guaranteed through suitable applications of widening[4] at recursive calls.

This approach operates over a worklist of $\Sigma^\sharp \times Loc$ pairs (i.e., a procedure-entry abstract state and a program location) and computes for each encountered (φ, ℓ) an abstract state $I(\varphi, \ell)$, which is best understood as the post-condition of a Hoare triple with pre-condition φ over the valid paths to ℓ from the entry of the containing procedure. Thus, when ℓ is a procedure exit location $\mathsf{exit}(\rho)$, the abstract state $I(\varphi, \ell)$ is equivalent to a procedure summary $\{\varphi\}\ \rho\ \{I(\varphi, \ell)\}$, which can be used to interpret calls to ρ in compatible abstract states.

4 DEMANDED SUMMARIZATION

In this section, we describe demanded summarization, which generates procedure summaries on demand using a tabulation-style interprocedural abstract interpretation. Demanded summarization bridges the gap between operational-style demanded abstract interpretation and denotational-style procedure summaries.

One can of course construct a single operational-style dependency structure over the interprocedural control-flow graph that results from connecting call sites $\ell \dashv\texttt{call}\ \rho\vdash\!\!\mapsto \ell'$ to procedures ρ (i.e., with ℓ transitioning to $\mathsf{entry}(\rho)$ and $\mathsf{exit}(\rho)$ to ℓ') with a suitable context abstraction such as k-limited call strings. However, as illustrated in Section 2, this global dependency structure induced by context-sensitive analysis is overly conservative for programs with good procedural abstraction and thus leads to poor incremental reuse.

Instead, we employ distinct intraprocedural dependency structures for each procedure summary, instantiated on demand. In particular, we assume that an initial DAIG $\mathcal{D}^{\mathsf{init}}_{\rho,\varphi}$ can be constructed for any procedure ρ with initial abstract state φ stored in cell $\underline{\mathsf{entry}(\rho)}$ and with otherwise uninitialized abstract-state cells; that is, $\mathcal{D}^{\mathsf{init}}_{\rho,\varphi}$ reifies the computational structure of analyzing procedure ρ, but with no actual analysis work performed yet beyond filling in the abstract state at $\mathsf{entry}(\rho)$.

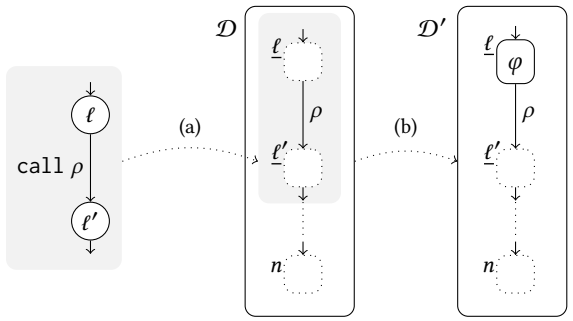
Although Stein et al. [2021a] give a mechanism to reify intraprocedural analysis computation, procedure call CFG edges must also be translated somehow. To do so, we extend slightly the DAIG syntax and semantics of Section 3.1 as follows. In Fig. 5, we show how a CFG edge $\ell \dashv\texttt{call}\ \rho\vdash\!\!\mapsto \ell'$ is encoded by a new kind of DAIG edge with label ρ connecting the abstract state reference cell $\underline{\ell}$ to that of $\underline{\ell'}$. This label ρ indicates that the function to compute $\underline{\ell'}$ from $\underline{\ell}$ is described by a summary of procedure ρ, which we can view as a (higher-order) reference to another DAIG for ρ. However, since the intraprocedural DAIG semantics of Stein et al. [2021a] have no means to evaluate such an edge, queries can now get "stuck" with a value for $\underline{\ell}$ but no way to compute a value for $\underline{\ell'}$.

Querying a DAIG $\mathcal{D}$ for the value of a cell n thus may result in either an abstract state to store in n (if no interprocedural analysis is required) or getting stuck and demanding a procedure summary.

To capture this second possibility – when intraprocedural analysis is unable to proceed without some procedure summary – we introduce a judgment form

$$\text{demanding a procedure summary} \quad \mathcal{D} \vdash n \overset{n'}{\rightsquigarrow} (\rho, \varphi)\ ;\ \mathcal{D}'$$

[4]Throughout this paper, we fix a strategy of widening at every recursive call, for simplicity and clarity of presentation. Our general approach is compatible with other widening strategies, too, but we choose not to introduce that degree of freedom into the formalism and instead present the simple and conservative widen-everywhere strategy. It is also not clear that arbitrary widening strategies give rise to from-scratch consistent implementations, if the choice of whether or not to widen depends on some information not preserved in the incremental setting.



Intraprocedural analysis demanding a summary for ρ: $\mathcal{D} \vdash n \overset{\underline{\ell'}}{\rightsquigarrow} (\rho, \varphi)$; $\mathcal{D}'$

Fig. 5. Translating and abstractly interpreting a procedure-call CFG edge. Dotted arrow (a) shows how a CFG edge of the form $\ell \dashv[\texttt{call } \rho]\mapsto \ell'$ is encoded into a corresponding DAIG edge labeled by ρ (both in shaded boxes). Intuitively, a procedure-labeled DAIG edge corresponds to computing a summary of procedure ρ by instantiating a DAIG for ρ as needed. Dotted arrow (b) shows the effect of a subsequent query for a cell named n that depends transitively on the value at ℓ'. The query for the value of n is blocked by needing to apply a summary for ρ, which is captured by the judgment shown above for demanding a summary.

which indicates that a query for the value named by n in $\mathcal{D}$ is stuck, unable to compute (and store at the return site n') the result of a call to ρ with abstract state φ and where $\mathcal{D}'$ reflects any intraprocedural analysis evaluation in $\mathcal{D}$ before it got stuck on the ρ-labeled edge.

Example 4.1. Fig. 5 gives a visual example of this judgment $\mathcal{D} \vdash n \overset{\underline{\ell'}}{\rightsquigarrow} (\rho, \varphi)$; $\mathcal{D}'$. The middle box, labeled by $\mathcal{D}$, shows a fragment of an initial DAIG containing a call to some procedure ρ, upon which some analysis state named n transitively depends. A query for the value of n triggers analysis computation (and caching of results) up to the cell $\underline{\ell}$ preceding the call site, at which it gets "stuck", needing a summary of ρ in initial state φ in order to compute an invariant at ℓ' and continue. The partially-analyzed state is reflected in DAIG $\mathcal{D}'$, shown in the right box.

In the rest of this section, we introduce *demanded summarization graphs* (DSGs), which enable these partial computations stuck on demanding a callee summary to get "unstuck." Evaluation of DSGs interleaves intraprocedural DAIG evaluation with procedure summary synthesis. Procedure summaries are synthesized by instantiating and/or evaluating DAIGs for procedures on demand.

A demanded summarization graph (DSG) $\mathcal{G} = \langle \mathcal{D}^*, \Delta \rangle$ consists of a DAIG for each partially- or fully-computed summary of a procedure ρ with initial abstract state φ, which we denote by $\mathcal{D}^*(\rho, \varphi)$. Thus, there may be multiple instantiated DAIGs for a procedure ρ, each with a different initial abstract state. Crucially for incremental analysis (cf. Section 4.2), it also maintains a demanded summarization dependency map Δ that records interprocedural analysis dependencies from applying procedure summaries.

A demanded summarization dependency δ has the form $(\rho', \varphi') \overset{n}{\hookleftarrow} (\rho, \varphi)$, which indicates that the return-site abstract state named by n in $\mathcal{D}^*(\rho', \varphi')$ depends upon the summary DAIG $\mathcal{D}^*(\rho, \varphi)$. Note that this is a *very fine-grained* dependency: not only do we record that procedure ρ' depends on procedure ρ, but also the relevant precondition states and the specific point n in the analysis of ρ' which depends on a summary of ρ.

We denote by $\mathcal{D}^{\mathcal{G}}_{\rho,\varphi}$ the DAIG $\mathcal{D}$ mapped to by (ρ, φ) in DSG $\mathcal{G}$. We use $\mathcal{G}[\mathcal{D}/(\rho, \varphi)]$ as shorthand for $\langle \mathcal{D}^*[\mathcal{D}/(\rho, \varphi)], \Delta \rangle$, that is, $\mathcal{G}$ with the DAIG summarizing ρ in initial state φ updated to $\mathcal{D}$.

$$
\begin{array}{rlll}
\text{demanded summarization graphs} & \mathcal{G} & ::= & \langle \mathcal{D}^*, \Delta \rangle \\
\text{intraprocedural analysis states} & \mathcal{D}^* & ::= & \varepsilon \mid \mathcal{D}^*; (\rho, \varphi) \mapsto \mathcal{D} \\
\text{summary dependency edges} & \delta \in \mathit{Dep} & ::= & (\rho', \varphi') \overset{n}{\hookleftarrow} (\rho, \varphi) \\
\text{summary dependency maps} & \Delta & ::= & \varepsilon \mid \Delta \,;\, \delta
\end{array}
$$

$$\boxed{\mathcal{G} \vdash \{\varphi\}\ \rho\ \{\varphi'\} \,;\, \mathcal{G}'}$$

$$\textsc{Summarize}\quad \frac{\mathcal{G} \vdash_{\varphi} \mathsf{exit}(\rho) \Downarrow \varphi' \,;\, \mathcal{G}'}{\mathcal{G} \vdash \{\varphi\}\ \rho\ \{\varphi'\} \,;\, \mathcal{G}'}$$

Fig. 6. A demanded summarization graph (DSG) $\mathcal{G}$ is a collection $\mathcal{D}^*$ of intraprocedural DAIGs overlaid by an interprocedural demanded summarization dependency map Δ. The demanded summarization dependency map Δ captures the essence of demanded summarization; it is progressively extended to capture the *dynamic* dependencies from using procedure summaries during demand-driven query evaluation that are later needed for incremental dirtying. The summarization judgment $\mathcal{G} \vdash \{\varphi\}\ \rho\ \{\varphi'\} \,;\, \mathcal{G}'$ makes explicit the interpretation of *operational* DAIG analysis results as *denotational* procedure summaries (i.e., Hoare triples over procedures).

Similarly, we denote by $\Delta_{\rho,\varphi}$ the set of dependencies in Δ on the procedure-ρ summary with initial state φ and use $\mathcal{G}[\delta]$ as shorthand for $\langle \mathcal{D}^*, \Delta; \delta \rangle$, that is, $\mathcal{G}$ with an added summary-dependency edge δ.

In Section 4.1, we define demand-driven *query* evaluation using the judgment form

$$\text{demanding an abstract state in a DSG} \quad \mathcal{G} \vdash_{\varphi} \ell \Downarrow \varphi' \,;\, \mathcal{G}'$$

that says, "Given a DSG $\mathcal{G}$, a query for the abstract state at ℓ under pre-condition φ for ℓ's enclosing procedure returns result φ' and updated DSG $\mathcal{G}'$."

Example 4.2 (DSG Queries). In the example of Fig. 3, analyzing the program after the edit corresponds to a derivation of the query evaluation judgment $\mathcal{G} \vdash_{\varphi} \ell \Downarrow \varphi' \,;\, \mathcal{G}'$, where:

- $\mathcal{G}$ is the DSG representing the analysis state immediately after the program edit is processed. $\mathcal{G}$ contains the two previously-computed summaries of `p` on the right side of the figure, and also the partially-computed summary of `main` after dirtying (see Section 3.1) from the edit.
- φ is the initial abstract state, φ' the final abstract state, and ℓ the exit location for `main`.
- $\mathcal{G}'$ is the fully-evaluated DSG after the query is complete. $\mathcal{G}'$ contains both previously-cached summaries of `p`, a now-fully-computed summary of `main`, and the freshly-computed summary of `p` (shown beneath the `main` summary).

Fig. 6 shows a $\mathcal{G} \vdash \{\varphi\}\ \rho\ \{\varphi'\} \,;\, \mathcal{G}'$ judgment form that makes explicit our interpretation of DAIG analysis results as composable procedure summaries. In particular, the Summarize rule shows how a composable procedure summary $\{\varphi\}\ \rho\ \{\varphi'\}$ (i.e., a Hoare triple) for the procedure ρ is implied by a DSG query result computed at the exit of the corresponding DAIG (i.e., $\mathcal{G} \vdash_{\varphi} \mathsf{exit}(\rho) \Downarrow \varphi' \,;\, \mathcal{G}'$). Intuitively, this rule captures reifying the denotational procedure summary through the operational fixed-point computation therein, applying local transfer functions and/or summary transformers.

4.1 Demand-Driven Query Evaluation in DSGs

Abstract interpretation in DSGs is *demand-driven* by default: analysis results are computed only as needed to answer queries. A *query* requests the abstract state at a specific program location ℓ

under some procedure-entry precondition φ, and may be issued directly by a developer through their IDE, programmatically by a client analysis, or internally in service of another query.

4.1.1 Queries. Queries in DSGs are governed by the $\mathcal{G} \vdash_{\varphi} \ell \Downarrow \varphi' ; \mathcal{G}'$ judgment form, introduced in the previous section and defined inductively by the Q-* (for "query") inference rules of Fig. 7.

We explain the inference rules in turn. The Q-Instantiate rule is used to instantiate a DAIG for a procedure on demand. More specifically, if the current DSG does not have a DAIG for the procedure ρ^{ℓ} containing ℓ with pre-condition φ (i.e., $(\rho^{\ell}, \varphi) \notin \mathrm{dom}(\mathcal{D}^*)$), Q-Instantiate instantiates a new DAIG $\mathcal{D}^{\mathsf{init}}_{\rho^{\ell},\varphi}$ for ρ^{ℓ}, with entry abstract state φ, and then re-issues the query for location ℓ in this extended DSG.

The Q-Delegate rule applies when the relevant DAIG is already available in the DSG and can handle the query on its own, i.e., without getting stuck due to dependence on another procedure summary. In this case, an intra-procedural DAIG query (discussed previously in Section 3.1) suffices to compute the result. Note that if the queried abstract state had previously been computed, no analysis computation is performed and the analysis state is unchanged.

Finally, the Q-Apply-Summary rule handles the case when a procedure summary must be applied to handle a query. This rule does the heavy lifting of composing analysis results by applying summaries and tracking interprocedural analysis dependencies. Its first premise states that the query-relevant DAIG $\mathcal{D}^{\mathcal{G}}_{\rho^{\ell},\varphi}$ is unable to produce a result by intraprocedural analysis, as the result transitively depends upon a call to ρ in abstract state φ' at the call site. The second premise uses an auxiliary summary query judgment form for

$$\text{computing and applying summaries} \quad \mathcal{G} \vdash (\rho, \varphi) \overset{n}{\rightsquigarrow} (\rho', \varphi') ; \mathcal{G}' \ .$$

It says, "In DSG $\mathcal{G}$ and in procedure ρ with pre-condition φ, a query at the return site n is resolved with a procedure summary over ρ' with pre-condition φ', yielding updated $\mathcal{G}'$." We discuss the rules for handling such summary queries in detail in Section 4.1.2.

Example 4.3 (Summary queries). In Fig. 3, each interprocedural dependency edge corresponds to a derivation of a summary query judgment $\mathcal{G} \vdash (\rho, \varphi) \overset{n}{\rightsquigarrow} (\rho', \varphi') ; \mathcal{G}'$. These summary queries arise as subderivations of a larger analysis query derivation, where

- $\mathcal{G}$ is the interprocedural analysis state in which intraprocedural analysis got stuck requiring a procedure summary to continue;
- ρ' is the callee function and φ' the entry abstract state of the summary at the edge's source;
- n is the name of the return-site abstract state of the callsite at the edge's destination;
- ρ is the caller function and φ the entry abstract state of the summary containing that callsite; and
- $\mathcal{G}'$ is the interprocedural analysis state extending $\mathcal{G}$ with any newly-computed summaries, and the return-site abstract state computed and stored at n.

The summary query judgment in Q-ApplySummary's premises demands a summary of the invoked procedure ρ with pre-condition φ', applying it to resolve the post-state n of a call in the summary DAIG indexed by (ρ^{ℓ}, φ). Once a compatible summary is applied for the call to ρ in the updated DSG $\mathcal{G}'$, the query for location ℓ is reissued, just like with Q-Instantiate.

Note the connection between stuck intraprocedural analysis (denoted with $\overset{n}{\rightsquigarrow}$) and interprocedural summary queries (denoted with $\overset{n}{\rightsquigarrow}$) in Q-Apply-Summary's premises – whenever analysis is unable to proceed without some procedure summary, this rule computes or fetches the requisite summary, applies it, and then analysis proceeds through the third premise.

4.1.2 Summary Queries. The complexity of interprocedural analysis thus lies in computing summaries with the judgment form $\mathcal{G} \vdash (\rho, \varphi) \overset{n}{\rightsquigarrow} (\rho', \varphi') ; \mathcal{G}'$ defined inductively with the SQ-* (for "summary query") rules of Fig. 7.

$\boxed{\mathcal{G} \vdash_\varphi \ell \Downarrow \varphi' ; \mathcal{G}'}$

Q-Instantiate

$$\frac{(\rho^\ell, \varphi) \notin \mathrm{dom}(\mathcal{D}^*) \qquad \langle \mathcal{D}^*; (\rho^\ell, \varphi) \mapsto \mathcal{D}^{\mathsf{init}}_{\rho^\ell,\varphi}, \Delta\rangle \vdash_\varphi \ell \Downarrow \varphi' ; \mathcal{G}}{\langle \mathcal{D}^*, \Delta\rangle \vdash_\varphi \ell \Downarrow \varphi' ; \mathcal{G}}$$

Q-Delegate

$$\frac{\mathcal{D}^{\mathcal{G}}_{\rho^\ell,\varphi} \vdash \underline{\ell} \Rightarrow \varphi' ; \mathcal{D} \qquad \mathcal{G}' = \mathcal{G}[\mathcal{D}/(\rho^\ell, \varphi)]}{\mathcal{G} \vdash_\varphi \ell \Downarrow \varphi' ; \mathcal{G}'}$$

Q-Apply-Summary

$$\frac{\mathcal{D}^{\mathcal{G}}_{\rho^\ell,\varphi} \vdash \underline{\ell} \overset{n}{\rightsquigarrow} (\rho, \varphi') ; \mathcal{D} \qquad \mathcal{G}[\mathcal{D}/(\rho^\ell, \varphi)] \vdash (\rho^\ell, \varphi) \overset{n}{\leftsquigarrow} (\rho, \varphi') ; \mathcal{G}' \qquad \mathcal{G}' \vdash_\varphi \ell \Downarrow \varphi'' ; \mathcal{G}''}{\mathcal{G} \vdash_\varphi \ell \Downarrow \varphi'' ; \mathcal{G}''}$$

$\boxed{\mathcal{G} \vdash (\rho, \varphi) \overset{n}{\leftsquigarrow} (\rho', \varphi') ; \mathcal{G}'}$

SQ-Other-Proc

$$\frac{\rho \neq \rho' \qquad \mathcal{G} \vdash \{\varphi'\}\rho'\{\varphi_{\mathsf{post}}\} ; \mathcal{G}' \qquad \mathcal{D}^{\mathcal{G}'}_{\rho,\varphi} \vdash n \Leftarrow \varphi_{\mathsf{post}} ; \mathcal{D} \qquad \mathcal{G}'' = \mathcal{G}'[\mathcal{D}/(\rho, \varphi)][(\rho, \varphi) \overset{n}{\hookleftarrow} (\rho', \varphi')]}{\mathcal{G} \vdash (\rho, \varphi) \overset{n}{\leftsquigarrow} (\rho', \varphi') ; \mathcal{G}''}$$

SQ-Other-Pre

$$\frac{\varphi' \not\sqsubseteq \varphi \qquad \mathcal{G} \vdash \{\varphi \nabla \varphi'\}\rho\{\varphi_{\mathsf{post}}\} ; \mathcal{G}' \qquad \mathcal{D}^{\mathcal{G}'}_{\rho,\varphi} \vdash n \Leftarrow \varphi_{\mathsf{post}} ; \mathcal{D} \qquad \mathcal{G}'' = \mathcal{G}'[\mathcal{D}/(\rho, \varphi)][(\rho, \varphi) \overset{n}{\hookleftarrow} (\rho, \varphi \nabla \varphi')]}{\mathcal{G} \vdash (\rho, \varphi) \overset{n}{\leftsquigarrow} (\rho, \varphi') ; \mathcal{G}''}$$

SQ-Self

$$\frac{\varphi' \sqsubseteq \varphi \qquad \mathcal{D}^{\mathcal{G}}_{\rho,\varphi} \vdash n \Leftarrow \bot ; \mathcal{D} \qquad \mathcal{G}[(\rho, \varphi) \overset{n}{\hookleftarrow} (\rho, \varphi)] \vdash \mathsf{fix}(\rho, \varphi); \mathcal{G}'}{\mathcal{G} \vdash (\rho, \varphi) \overset{n}{\leftsquigarrow} (\rho, \varphi') ; \mathcal{G}'}$$

$\boxed{\mathcal{G} \vdash \mathsf{fix}(\rho, \varphi); \mathcal{G}'}$

F-Converge

$$\frac{\mathcal{G} \vdash \{\varphi\}\ \rho\ \{\varphi'\} ; \mathcal{G}' \qquad \mathrm{R}_{\rho,\varphi}(\mathcal{G}') = \emptyset}{\mathcal{G} \vdash \mathsf{fix}(\rho, \varphi); \mathcal{G}'}$$

F-Step

$$\frac{\mathcal{G} \vdash \{\varphi\}\ \rho\ \{\varphi'\} ; \mathcal{G}' \qquad \{(n_1, \varphi_1) \ldots, (n_k, \varphi_k)\} = \mathrm{R}_{\rho,\varphi}(\mathcal{G}') \neq \emptyset \qquad \mathcal{D}_0 = \mathcal{D}^{\mathcal{G}'}_{\rho,\varphi} \quad \mathcal{D}_{i-1} \vdash n_i \Leftarrow \varphi_i \nabla \varphi' ; \mathcal{D}_i \quad \text{for } 1 \leq i \leq k \qquad \mathcal{G}'[\mathcal{D}_k/(\rho, \varphi)] \vdash \mathsf{fix}(\rho, \varphi); \mathcal{G}''}{\mathcal{G} \vdash \mathsf{fix}(\rho, \varphi); \mathcal{G}''}$$

$$\text{where } \mathrm{R}_{\rho,\varphi}(\langle \mathcal{D}^*; (\rho, \varphi) \mapsto \mathcal{D}, \Delta\rangle) = \left\{ (n\ ,\ \mathcal{D}(n)) \;\middle|\; \begin{array}{l} (\rho, \varphi) \overset{n}{\hookleftarrow} (\rho, \varphi) \in \Delta \\ \wedge\ \mathcal{D}(\underline{\mathsf{exit}(\rho)}) \not\sqsubseteq \mathcal{D}(n) \end{array} \right\}$$

Fig. 7. Operational semantics rules governing analysis computation in DSGs. The *query* judgment form $\mathcal{G} \vdash_\varphi \ell \Downarrow \varphi' ; \mathcal{G}'$ is read as "a query against DSG $\mathcal{G}$ for the abstract state at location ℓ with procedure-entry abstract state φ yields result φ' and updated DSG $\mathcal{G}'$."
It is defined via mutual recursion with a *summary query* judgment form $\mathcal{G} \vdash (\rho, \varphi) \overset{n}{\leftsquigarrow} (\rho', \varphi') ; \mathcal{G}'$ for demanding and applying summaries, and a *fixed-point* judgment form $\mathcal{G} \vdash \mathsf{fix}(\rho, \varphi); \mathcal{G}'$ for fixed-point computations on self-referential summaries of recursive procedures.
The shorthand $\mathrm{R}_{\rho,\varphi}(\mathcal{G})$ describes the set of recursive-call return sites that do *not* over-approximate the procedure-exit abstract state in the DAIG summarizing ρ under pre-condition φ .

Let us first consider the simpler case of a non-recursive call to a different procedure ρ' from the current caller ρ with an abstract state φ' at the call site, handled by the SQ-Other-Proc rule. In the second premise, we compute a Hoare-style summary $\mathcal{G} \vdash \{\varphi'\}\ \rho'\ \{\varphi_{\text{post}}\}\ ; \mathcal{G}'$ of the callee; since these judgment forms are mutually inductively defined, deriving this second premise can involve arbitrary additional analysis computations.

Then, we write φ_{post} to the return site n in the third premise and add the fine-grained demanded summarization dependency $(\rho, \varphi) \overset{n}{\longleftarrow} (\rho', \varphi')$ to $\mathcal{G}''$, expressing that the value at the return site n now depends on the $\{\varphi'\}\ \rho'\ \{\varphi_{\text{post}}\}$ summary. Critically, if the applied summary is ever invalidated by incremental edits, then the return site n must also be invalidated.

The next two rules SQ-Other-Pre and SQ-Self implement an incremental and demand-driven tabulation with (directly[5]) recursive procedures. Note that both rules derive judgments in which a query in (ρ, φ) depends on some summary of the same procedure ρ.

In the case where the callsite state φ' is *not* included in the pre-condition of this procedure DAIG φ (i.e., $\varphi' \not\sqsubseteq \varphi$) in SQ-Other-Pre, we demand another summary of ρ with pre-condition $\varphi \nabla \varphi'$, which yields post-condition φ_{post}, allowing analysis to proceed in the same manner as in the non-recursive case with SQ-Other-Proc. This summary is guaranteed to be compatible (i.e., $\varphi' \sqsubseteq \varphi \nabla \varphi'$), but we need to apply widening in the abstract pre-condition to ensure that this iterative demanding of new summaries for procedure ρ converges. Intuitively, applying SQ-Other-Pre corresponds to a demanded unrolling of recursive calls a finite (but *a priori* unbounded) number of times determined by the widening operator ∇.

In the other case (i.e., the SQ-Self rule), the summary of procedure ρ that we need is the same one that we are currently demanding — a *self-referential summary*. That is, the recursive-call–site state φ' is contained in the pre-condition φ of this summary that is currently being demanded (i.e., $\varphi' \sqsubseteq \varphi$). When a procedure summary depends on itself, some special care must be taken to compute a fixed point along the control-flow cycle(s) formed between the procedure exit and recursive return site(s), as illustrated in Fig. 4. This fixed-point computation is implemented by a judgment form for

$$\text{demanding a self-referential summary} \quad \mathcal{G} \vdash \mathsf{fix}(\rho, \varphi); \mathcal{G}' \ ,$$

which says, "In DSG $\mathcal{G}$, the fixed point of a self-referential summary of procedure ρ with pre-condition φ yields an updated DSG $\mathcal{G}'$."

Note that the summarization, abstract state query, summary resolution, and self-referential summary fixed-point iteration judgments are mutually recursively defined, so any number of summaries can be produced, cached and/or memo-matched upon in the process.

4.1.3 Fixed-points. The SQ-Self rule initializes this fixed-point iteration, which is then implemented by the F-* (for "fixed-point") rules. Since analysis has yet to reach the procedure exit, we initialize the call's return state at n with $\bot$ in the second premise and then demand a fixed-point for this self-referential summary (ρ, φ) in the third. Note that this self dependency is made explicit with the self-referential demanded summarization dependency $(\rho, \varphi) \overset{n}{\longleftarrow} (\rho, \varphi)$ extending $\mathcal{G}$.

The fixed-point computation of a self-referential procedure summary can now be described using the machinery defined to this point. At each step, we demand a summary for procedure ρ with pre-condition φ, that is, $\mathcal{G} \vdash \{\varphi\}\ \rho\ \{\varphi'\}\ ; \mathcal{G}'$. The shorthand $\mathsf{R}_{\rho,\varphi}(\mathcal{G}')$ then yields the set of recursive return sites whose abstract state is not over-approximated by the procedure-exit state φ' in the procedure DAIG indexed by (ρ, φ). When this set is empty (F-Converge), the fixed-point iteration has converged. Otherwise, some return sites n_i have abstract state φ_i not over-approximated by the procedure-exit state φ', so F-Step applies, widening φ' onto each return-site state before taking

[5]Note that we assume that there is no mutual recursion (e.g., each strongly-connected component in the call graph is compiled to a directly recursive procedure).

another step in the fixed-point iteration. Importantly, the widened state $\varphi_i \nabla \varphi'$ is written to cell n using the dirtying-on-an-edit judgment, meaning forward-reachable nodes from n in the procedure DAIG get dirtied.

Example 4.4 (Fixed-points). As a concrete example, such a fixed-point computation is instantiated in Fig. 4, showing the steps taken to produce a procedure summary of p using Summarize.

In the initial (left-most) state, Q-Apply-Summary is needed because intraprocedural analysis is stuck at a callsite. Then, each transition in the figure applies the DSG evaluation semantics as follows:

(a) To satisfy Q-Apply-Summary's second antecedent – i.e. to apply a procedure summary – we must derive one of the SQ-* rules; in this case SQ-Self applies, since the callee and caller are the same and the callsite abstract state is smaller than the caller initial state. SQ-Self writes $\bot$ to the return site, then continues analysis (via the summarization judgment in the first antecedent of both F-* rules) to the procedure exit in order to derive a fixed-point judgment.
(b) Since the analysis hasn't yet converged (i.e. $R_{\rho,\varphi}(\mathcal{G}')$ is non-empty, since the $\bot$ abstract state at the return site is smaller than the exit state), F-Step applies. The exit state is widened into the return site, and another fixed-point judgment must be derived to satisfy the final antecedent so analysis runs to the procedure exit as before.
(c) Since analysis still hasn't converged, F-Step applies again, analogously to step (b). At this point, the return site abstract state is larger than the exit state, so F-Converge applies and the derivation is complete.

Note that the "steps" here are not sequential composition in the style of small-step operational semantics but rather nested derivations in the style of big-step operational semantics; each step is a sub-derivation of the prior using the semantics of Fig. 7.

This example demonstrates the interplay between the three judgment forms governing analysis queries in DSGs: to summarize a procedure, we query for the procedure-exit abstract state; in order to get to the procedure exit, we must derive procedure summaries; when those summaries are self-referential, we must derive a fixed-point.

Note that these interleavings can be far more complex in general: some (possibly recursive) callee procedure summaries may be computed as further subderivations of each judgment form.

4.2 Incremental Edits in DSGs

When the program under analysis is edited, a DSG must discard those analysis results that are potentially affected while retaining as many cached results as possible for future incremental reuse. This operation is given by an operational semantics over analysis states $\mathcal{G}$, just as with demand-driven query evaluation. The judgment form $\mathcal{G} \vdash_\rho n \Leftarrow s \,;\, \mathcal{G}'$ of Fig. 8 defines the impact of an edit that modifies the statement named by n in procedure ρ, discarding facts from DSG $\mathcal{G}$ to yield $\mathcal{G}'$. Note that although this judgment as-written only applies to CFG-structure-preserving statement *modifications*, the extension to statement *insertions* and *deletions* is straightforward: we insert or remove the DAIG region corresponding to the edit, then dirty $\mathcal{G}$ from its exit (merging the entry and exit location of deleted regions).

It is also important to notice that no special handling of edits involving procedure calls is needed – the transitive dirtying operation is based on the summary dependencies of the *pre-edit* program, as recorded in Δ, and any relevant changes to the program's call structure will be reflected in summary dependencies the next time the procedure is analyzed. This is not just an optimization; it is crucial for correct and from-scratch consistent demanded analysis.

The top-level program edit rule E-Delegate is required since there may be multiple extant summary DAIGs for the edited procedure ρ, each with its own entry abstract state. E-Delegate

$$\boxed{\mathcal{G} \vdash_{\rho} n \Leftarrow s \,;\, \mathcal{G}'}$$

E-Delegate

$$\frac{\mathcal{G}_0 = \langle \mathcal{D}^*, \Delta \rangle \qquad \{\varphi_1, \ldots, \varphi_k\} = \{\varphi | (\rho, \varphi) \in \mathsf{dom}(\mathcal{D}^*)\} \qquad \mathcal{G}_{i-1} \vdash_{\rho,\varphi_i} n \Leftarrow s \,;\, \mathcal{G}_i \text{ for } 1 \leq i \leq k}{\mathcal{G}_0 \vdash_{\rho} n \Leftarrow s \,;\, \mathcal{G}_k}$$

$$\boxed{\mathcal{G} \vdash_{\rho,\varphi} n \Leftarrow s_\varepsilon \,;\, \mathcal{G}'}$$

D-DemandedSummaries

$$\frac{\langle D^*, \Delta \rangle \vdash_{\rho',\varphi'} n' \Leftarrow \varepsilon \,;\, \mathcal{G} \qquad \mathcal{G} \vdash_{\rho,\varphi} n \Leftarrow s_\varepsilon \,;\, \mathcal{G}'}{\langle \mathcal{D}^*, \Delta; (\rho', \varphi') \overset{n'}{\hookleftarrow} (\rho, \varphi) \rangle \vdash_{\rho,\varphi} n \Leftarrow s_\varepsilon \,;\, \mathcal{G}'}$$

D-DAIG

$$\frac{\Delta_{\rho,\varphi} = \emptyset \qquad \mathcal{D}^*(\rho, \varphi) \vdash n \Leftarrow s_\varepsilon \,;\, \mathcal{D}' \qquad \Delta' = \left\{ \delta \;\middle|\; (\rho, \varphi) \overset{n'}{\hookleftarrow} (\rho', \varphi') = \delta \in \Delta \;\wedge\; \mathcal{D}'(n') = \varepsilon \right\}}{\langle \mathcal{D}^*, \Delta \rangle \vdash_{\rho,\varphi} n \Leftarrow s_\varepsilon \,;\, \langle \mathcal{D}^*[\mathcal{D}'/(\rho, \varphi)], \Delta \backslash \Delta' \rangle}$$

Fig. 8. Operational semantics rules governing *edits* to a program under analysis. The *program-edit* judgment form $\mathcal{G} \vdash_{\rho} n \Leftarrow s \,;\, \mathcal{G}'$ is read as "$\mathcal{G}$ is updated to $\mathcal{G}'$ by an edit that writes statement s at the position named by n in procedure ρ", and is defined in terms of the *dirtying* judgment form $\mathcal{G} \vdash_{\rho,\varphi} n \Leftarrow s_\varepsilon \,;\, \mathcal{G}'$, which applies edits or propagates changes across demanded summarization dependencies.

simply delegates the dirtying of each such DAIG to the D-* rules in Figure 8, using judgment form $\mathcal{G} \vdash_{\rho,\varphi} n \Leftarrow s_\varepsilon \,;\, \mathcal{G}'$ to dirty a specific DAIG.

Rule D-DemandedSummaries is the inductive case for dirtying demanded summarization dependencies: when a fact named by n' in the ρ', φ' DAIG depends upon the ρ, φ DAIG we are currently dirtying, we drop the dependency edge, dirty from n' in the ρ', φ' DAIG, and continue recursively. Once all demanded summarization dependencies have been processed in that manner, the D-DAIG base case can be applied, dirtying any affected analysis results in the relevant DAIG (and their dependency edges Δ').

Formalizing incremental and demand-driven interprocedural analysis in this manner exposes questions of what is minimal dirtying and maximal incremental reuse. The analysis semantics we have described and implemented in this section dirty all (transitive) callers of any edited procedure, as those analysis results depend on a summary which has been dirtied. This is optimal in the sense that it dirties exactly those results potentially invalidated by an edit, and our query semantics are optimal in the sense that they lazily compute only what is needed to resolve a query.

However, those callers depend on the summary triple only, not the underlying procedure implementation. As such, it is possible that by re-demanding the exit location of the callee, a DSG may discover that a particular edit does not change the summary post-condition and thus dirtying need not propagate to callers. On the other hand, this eager re-demanding may not have been needed to respond to any future query.

5 FROM-SCRATCH CONSISTENCY

We now define and give proof sketches for several critical meta-theoretic properties of demanded summarization graphs, namely *termination* of queries and edits, *from-scratch consistency* with the underlying batch abstract interpreter, and *soundness* with respect to the concrete semantics. Proofs are elided here, but can be found in Appendix A.

Definition 5.1 (DSG Semantic Consistency). We say that a DSG $\mathcal{G} = \langle \mathcal{D}^*, \Delta \rangle$ is *semantically consistent* when it is syntactically well-formed and consistent with the program structure and underlying abstract interpretation semantics.

- Each constituent DAIG $\mathcal{D}^*(\rho, \varphi)$ is well-formed and consistent with the corresponding procedure CFG $P(\rho)$ and intraprocedural abstract semantics $\langle \Sigma^\sharp, \varphi, [\![\cdot]\!]^\sharp, \sqsubseteq, \sqcup, \nabla \rangle$.
- Each return-site abstract state has a corresponding dependency edge from its callee in Δ. That is, if n names a non-empty ref cell in some $\mathcal{D} = \mathcal{D}^*(\rho, \varphi)$ with a ρ'-labeled edge to n from n', then either
  - $(\rho, \varphi) \overset{n}{\hookleftarrow} (\rho', \mathcal{D}(n')) \in \Delta$ (when $\rho \neq \rho'$) or
  - $(\rho, \varphi) \overset{n}{\hookleftarrow} (\rho', \varphi \nabla \mathcal{D}(n')) \in \Delta$ (when $\rho = \rho'$).
- Dependency edges $(\rho', \varphi') \overset{n}{\hookleftarrow} (\rho, \varphi)$ in Δ are consistent with the relevant DAIGs, in that
  - $\mathcal{D}^*(\rho', \varphi')(n) = \mathcal{D}^*(\rho, \varphi)(\mathsf{exit}(\rho))$ (when $\rho \neq \rho'$) or
  - $\mathcal{D}^*(\rho', \varphi')(n) \sqsupseteq \mathcal{D}^*(\rho, \varphi)(\mathsf{exit}(\rho))$ (when $\rho = \rho'$).
- Analysis results in DAIGs are equal to the corresponding invariants produced by batch tabulation, where the domains coincide: for all $\mathcal{D}^*(\rho, \varphi)(\underline{\ell})$ where $(\varphi, \ell) \in \mathsf{dom}(I)$, $\mathcal{D}^{\mathcal{G}}_{\rho,\varphi}(\underline{\ell}) = I(\varphi, \ell)$.

Lemma 5.1 (Consistency Preservation). *If $\mathcal{G}$ is semantically consistent (with respect to a program P) then:*
if $\mathcal{G} \vdash_\varphi \ell \Downarrow \varphi' ; \mathcal{G}'$ then $\mathcal{G}'$ is semantically consistent, and
if $\mathcal{G} \vdash_\rho n \Leftarrow s ; \mathcal{G}'$ then $\mathcal{G}'$ is semantically consistent (with respect to the edited version of P).

Preservation of each of the four conditions of semantic consistency can be shown by induction on the derivations of the query and edit judgments, and ensures that the theorems to follow are applicable to the sequences of queries and edits made during an interactive analysis session.

Theorem 5.2 (Termination). *Queries and edits terminate:*

- *For all $\varphi, \ell \in L$, and consistent $\mathcal{G}$, there exist φ' and $\mathcal{G}'$ such that $\mathcal{G} \vdash_\varphi \ell \Downarrow \varphi' ; \mathcal{G}'$.*
- *For all s, n, ρ, and consistent $\mathcal{G}$ where n names a CFG edge in E^ρ, there exists a $\mathcal{G}'$ such that $\mathcal{G} \vdash_\rho n \Leftarrow s ; \mathcal{G}'$.*

In other words, any query or edit against a consistent DSG has a finite corresponding big-step evaluation. The edit-termination condition (the second bullet point) is straightforward: Δ and k are both finite in E-Delegate because $\mathcal{G}$ is consistent, and each of its premises terminates because the only recursive dirtying rule D-DemandedSummaries decreases in Δ.

The query-termination condition is more complicated, but can be shown by induction on the number of transitive callees of the procedure ρ^ℓ in which the query is issued, which decreases at all non-recursive calls. At recursive calls, termination relies on the convergence of the underlying abstract interpreter's widening operator.

Theorem 5.3 (From-Scratch Consistency). *Query results are equal to the corresponding invariant computed by tabulation: if $\mathcal{G}$ is semantically consistent, $\mathcal{G} \vdash_\varphi \ell \Downarrow \varphi' ; \mathcal{G}'$, and $(\varphi, \ell) \in \mathsf{dom}(I)$ then $\varphi' = I(\varphi, \ell)$.*

Intuitively, query results are from-scratch consistent due to (1) the from-scratch consistency of intraprocedural analysis in DAIGs, and (2) the fact that summary triples derived by $\mathcal{G} \vdash \{\varphi\}\, \rho\, \{\varphi'\} ; \mathcal{G}'$ are identical to the summary edges produced by a batch tabulation and are applied at call sites only when their pre-condition matches the caller abstract state.

Corollary 5.4 (Soundness). *Query results are sound with respect to the concrete semantics: If $\mathcal{G} \vdash \ell \Downarrow \varphi ; \mathcal{G}'$ and $\mathcal{G}$ is semantically consistent then $\sigma \models \varphi$ for all $\sigma \in [\![\ell]\!]^*$*

Soundness is a strictly weaker condition than from-scratch consistency: since demanded query results are the same as would be computed from-scratch by the batch analysis that is sound, so too are demanded query results.

6 IMPLEMENTATION & EVALUATION

In this section, we describe a prototype analysis framework based on DSGs and evaluate it on synthetically generated benchmarks as well as a corpus of bug-fix program edits drawn from open-source Java applications.

A core challenge in empirically evaluating an incremental or demand-driven analysis is the dearth of publicly-available data on real-world program edits and analysis queries.

Although some previous research on incremental and demand-driven program analysis has demonstrated considerable performance benefits, it has typically been applied to carefully restricted programming languages or synthetically-generated program edits [Stein et al. 2021a; Szabó et al. 2021].[6] Our aim is to show that demanded summarization can provide comparable performance benefits, while also supporting interprocedural analysis, procedure summarization, recursion, and the associated complexities.

On the other hand, some existing work [Arzt and Bodden 2014; Erhard et al. 2022] has been evaluated on open-source commit histories, providing stronger evidence of applicability to real edits but making it significantly more difficult to meaningfully evaluate interactive performance since there are fewer data points and the edits are of much coarser granularity.

Our Approach. We aim to cover both bases using two distinct sets of benchmarks: a synthetic corpus (Section 6.2) is designed to evaluate the *scalability* and efficacy of the demanded summarization algorithm, while an open-source corpus (Section 6.3) demonstrates the *generalizability* of these results to more realistic programs and edits and the practicability of incremental analysis infrastructure. Taken together, these experiments demonstrate the promise of demanded summarization as a framework for reliable interactive static analysis tools.

Since this paper's contribution is a domain-generic framework, we choose standard abstract domains throughout this section; our aim is to evaluate the scalability and generalizability of the technique rather than the particulars of any given abstract domain. Nonetheless, we instantiate the framework with several different domains – including some with infinite height and non-monotonic widening operators – throughout the evaluation so as to demonstrate its genericity.

6.1 Implementation

We implemented a prototype analysis framework in approximately 7000 lines of OCaml code, split roughly evenly between frontend infrastructure and analysis logic. The framework's frontend, intermediate representations, and core analysis engine are all designed to support incremental edits and demand queries. The implementation and experimental setup are publicly available on Github [Stein et al. 2021b].

Frontend. We use the tree-sitter incremental parsing library to interpret source-level changes at the granularity of concrete syntax tree nodes [Brunsfeld 2021]. In practice, we consider program edits that add or delete procedures or modify their headers; add, modify, or delete statements; and modify the headers of loops and conditionals. These concrete syntax tree edit scripts can then be interpreted in-place on our control-flow graph IR and analysis data structures.

[6] Notably, every program in the corpus of Section 6.3 includes some recursive procedure(s) and is thus out of reach of of Stein et al. [2021a]'s operational DAIG approach, and both Stein et al. [2021a] and Szabó et al. [2021] are evaluated on synthetic sequences of edits.

In order to resolve virtual calls, we rely on an upfront application-only callgraph computed using the WALA [WALA 2021] analysis library. A virtual call with multiple potential targets is interpreted as non-deterministic choice over direct calls to each target.

WALA does not provide incremental callgraphs, but techniques (e.g., [Schubert et al. 2021]) exist for incremental callgraph construction, which we could adopt in the future. Our use of an upfront callgraph is meant simply to exclude virtual call resolution issues from these experiments, since it is largely orthogonal to this paper's core contributions on abstract interpretation and dataflow analysis.

Analysis. The core analysis engine of our framework is a fairly direct implementation of the demanded summarization graph $\mathcal{G}$: we keep a map from procedure identifiers and procedure-entry abstract states to DAIGs, which we issue queries against, instantiate, and produce summaries for as needed to respond to extrinsically-provided queries.

Our intermediate representation of programs is similar to the control-flow graph language P of Section 3, the main difference being support of formal/actual parameter binding and variable scoping. These details are elided from the formalism for the sake of simplicity and clarity, but can in practice be handled using standard techniques without any major change to the underlying system as formalized in this paper.

The analysis engine is domain-agnostic: it is parameterized over an abstract domain module which provides standard abstract domain operations (essentially $\langle \Sigma^\sharp, \varphi_0, [\![\cdot]\!]^\sharp, \sqsubseteq, \sqcup, \nabla \rangle$) but can be implemented in a general-purpose language and needs not be aware of incrementality or demand.

We have instantiated the implementation with interval and octagon domains based on the APRON numerical domain library [Jeannet and Miné 2009] and a simple list-segment shape analysis domain, all of which are of infinite height and have non-monotonic widenings. To better exercise our analysis on object oriented programs in Section 6.3, we also implemented a finite domain tracking nullability of variables and heap addresses, using allocation sites to abstract memory locations.

Throughout this section, we consider four variants of the demanded summarization algorithm to re-analyze each program after an edit:

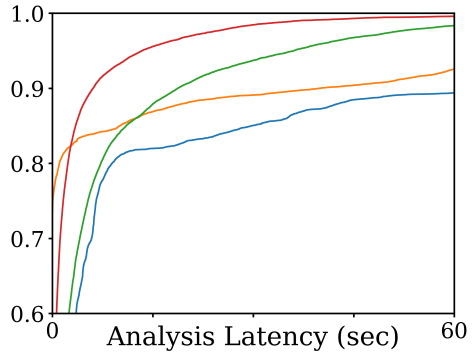
- *Batch* analysis, wherein the program is analyzed exhaustively from scratch when the program changes;
- *Incremental* analysis, which analyzes the program exhaustively and eagerly when the program changes, but reuses results from the previous version where possible;
- *Demand-Driven* analysis, which computes only those summaries that are required to respond to a client-issued query, but discards all analysis state when the program changes; and
- *Demanded* (i.e., incremental and demand-driven) analysis, which both reuses previous-version results and also computes only those summaries that are required to respond to queries.

The four variants amount to configuration options in our analysis framework: essentially, the incremental analyses apply the edit semantics of Fig. 8 while the demand-driven analyses apply the query semantics of Fig. 7.

6.2 Scalability: Synthetic Benchmarks

In order to evaluate the scalability and responsiveness of the demanded summarization approach, we analyze a synthetic workload of program edits and queries in each of the four configurations described above.

We instantiate the analysis framework for these experiments with an octagon domain backed by the APRON library [Jeannet and Miné 2009]; octagons are a popular relational numerical abstract domain, and APRON is a widely used implementation and thus has performance characteristics



	Analysis Time (sec)			
	mean	p50	p95	p99
Batch	16.4	1.1	118.8	149.3
Incremental	9.5	0.1	79.0	116.0
Demand-driven	6.9	0.3	32.8	130.9
Demanded	2.7	<0.1	10.3	36.6

Fig. 9. Summary-based octagon analysis run times on the synthetic benchmarks described in Section 6.2. The cumulative distribution plot shows the fraction of analysis runs (y axis) completed by each analysis configuration within some time interval (x axis): ordered from top to bottom at the right of the plot, the lines represent *demanded*, *demand-driven*, *incremental*, and *batch* analysis. The table shows corresponding summary statistics for each configuration, including the mean, median, 95th percentile, and 99th percentile analysis costs.

representative of many real-world domains. Furthermore, the infinite height lattice and non-monotonic widening operator of the octagon domain render it out of the reach of most existing incremental or demand-driven analysis techniques.

Programs are generated in an imperative language with arithmetic, booleans, conditional branching, while-loops, and recursive (but not mutually recursive) procedure calls. Each edit selects a random program location at which to add a loop, procedure call, conditional, or statement with probability 5%, 10%, 10%, and 75% respectively; statements and expressions are generated probabilistically from their respective grammars, and procedure call edits introduce a new procedure as their target 5% of the time.

These hyperparameters are chosen to simulate the *structure* of typical programs and edits thereof, allowing us to evaluate the performance characteristics of each analysis configuration with much more data and at a larger scale than is possible with real code and human subjects. Despite our efforts to produce realistically-structured synthetic programs, possible disparity with real-world control-flow structure is an inevitable threat to validity in these experiments. However, we argue that the statement-language abstract semantics and domains are less critical, acting as a constant factor throughout but not affecting the bottom-line comparisons between analysis configurations.

We analyze 8 separate trials of 2500 edits in each configuration, beginning with a program consisting only of a no-op `main` procedure and seeding the random number generators such that the same edits are analyzed by each analysis variant. Given the probabilistic grammar described above, this produces programs that are roughly 3000 significant lines of code on average at the end of each trial. In the demand-driven configurations, we issue three random queries between each edit.

The results are shown in Fig. 9 in the form of a continuous distribution plot and a table of summary statistics. Both incremental analysis and demand-driven analysis offer a significant improvement over the batch analysis baseline, both in terms of their average analysis cost and worst-case runtime at the 95th and 99th percentiles. However, demanded summarization, by combining incremental and demand-driven techniques, yields analysis results in 2.7 seconds on average and significantly mitigates the worst-case analysis costs at the tail of the distribution.

Of course, this performance improvement comes at the cost of an increased memory footprint. Profiling the memory usage of each configuration across the experiments, we find sub-linear increases for batch and demand-driven analysis, up to 450 and 347 MB respectively after 2500 edit/analysis iterations. Since both configurations discard all analysis results every iteration, this total memory footprint is roughly what is needed to analyze the full program (batch) and the transitive dependencies of a random query (demand-driven).

On the other hand, both the incremental and demanded configurations consume memory approximately linear in the size of the edit history, growing to 11.1 and 24.1 GB respectively after 2500 edit/analysis iterations. Though this is too large for most local development environments, it is feasible in a CI environment and could be tuned to hardware constraints using the memory-saving optimizations described in Section 7.2 without sacrificing the formal guarantees of demanded summarization.

As a whole, these experiments demonstrate that the combination of incremental and demand-driven techniques leveraged by demanded summarization is well-suited to interactive use, even in this setting of tabulation-based interprocedural analysis over recursive procedures.

6.3 Generalizability: Open-Source Corpus

We run our prototype analysis framework on a subset of the BugSwarm dataset, which consists of program *pairs* drawn from open-source applications and their continuous integration histories [Tomassi et al. 2019]. Each program pair consists of a "fail" version in which a CI failure was observed, and a subsequent "pass" version in which the failure is corrected.

We restrict our attention to pairs where the edit is in application code (rather than configuration or tests), affects between 1 and 200 lines of code, and the failure is not a compilation failure. On average, each program is 45 kLOC and consists of around 5000 distinct procedures. They make extensive use of Java language features including exceptions and a wide variety of control-flow mechanisms, but we do not consider programs that make use of lambda expressions or method references.

Table 1 shows the result of analyzing ten Java program pairs from the BugSwarm dataset using the interval analysis and nullability analysis described in Section 6.1. The interval abstract domain is widely used in practice for numerical analysis problems, but is difficult to handle incrementally due to its infinite height and non-monotonic widening operator, while the nullability abstract domain is finite but more well-suited to the analysis of the object oriented programs found in the data set. For the demand-driven configurations, we select query locations that correspond to the failures observed in the initial program version or as near as possible in our internal representation. These demand queries aim to emulate the natural use-case of querying at the alarm location after making an edit to address the alarm, for example to enable interactive and responsive IDE features built on top of an abstract interpreter.

Note that we do not purport to statically identify each of the failures and prove their fixes correct; the bugs are varied and complex, and the design of domains to reason precisely about their semantics is outside the scope of this paper. We use the BugSwarm corpus as a source of *real-world* bug-fixing edits, investigating the potential analysis cost savings that can be realized by demanded summarization in realistically-structured programs under actual patterns of queries and edits.

The analysis results are shown in Table 1, which presents symbolic metrics of analysis cost in terms of abstract states computed during reanalysis, in addition to wall-clock time and average peak memory usage.

We see that batch analysis requires computing tens to hundreds of thousands of abstract states for each program edit. Both incremental analysis and demand-driven analysis represent significant improvements over the batch approach, but exhibit high variability: depending on the location and

Table 1. Statistics about programs, edits, and analysis thereof in the open-source corpus, showing the relative degrees of reuse that are achieved by each analysis configuration. Artifacts refer to BugSwarm program pairs, for which we report final program size (in kLOC), program edit size (eLOC), and application-only callgraph size (|CG|).
We analyze each program pair in three domains: the interval (Itv.), octagon (Oct.), and nullability domains (N) described in Section 6.1. Then, we report for each analysis configuration the number of abstract states computed during reanalysis after applying the edit, in raw terms (#φ) for batch analysis and as a percentage (%φ) of the batch analysis baseline for each other configuration, as well as the amount of time required.
Lastly, we report the average peak memory usage of the analysis for each analysis configuration.

					Batch		Incremental		Dem.-Driven		Demanded	
artifact	kLOC	eLOC	|CG|	Domain	#φ	(s)	%φ	(s)	%φ	(s)	%φ	(s)
				Itv.	7029	0.20	12.5	0.02	28.0	0.10	0.2	<0.01
2021a	32.3	4	1233	Oct.	6441	0.31	4.7	0.02	24.4	0.11	7.3	0.11
				Null	13972	0.20	7.6	0.01	16.6	0.05	1.0	<0.01
				Itv.	12199	0.25	1.1	0.01	0.3	<0.01	0.0	<0.01
2021b	147.6	56	2920	Oct.	12495	0.40	1.0	0.01	0.3	<0.01	0.0	<0.01
				Null	19558	0.24	0.7	<0.01	0.2	<0.01	0.0	<0.01
				Itv.	17084	0.40	0.2	<0.01	48.7	0.27	<0.1	<0.01
2021c	15.8	4	1599	Oct.	17165	0.74	0.3	0.03	48.7	0.54	0.2	0.01
				Null	88160	2.61	<0.1	0.01	31.2	0.84	<0.1	<0.01
				Itv.	35395	1.35	<0.1	0.02	0.0	<0.01	0.0	<0.01
2021d	45.7	134	3866	Oct.	37257	3.20	<0.1	0.03	0.0	<0.01	0.0	<0.01
				Null	348760	16.59	<0.1	<0.01	0.0	<0.01	0.0	<0.01
				Itv.	73322	6.88	<0.1	0.03	3.4	0.09	<0.1	<0.01
2021e	36.2	8	7379	Oct.	74026	29.86	0.1	0.12	3.2	0.17	<0.1	<0.01
				Null	542853	29.01	<0.1	0.04	0.6	0.08	<0.1	<0.01
				Itv.	77954	6.94	0.1	0.04	5.6	0.23	0.0	<0.01
2021f	39.4	4	8212	Oct.	79909	40.63	0.1	0.06	3.2	0.18	0.0	0.01
				Null	622602	33.71	1.5	5.91	0.8	0.12	<0.1	0.02
				Itv.	78906	6.57	<0.1	0.03	2.4	0.07	<0.1	<0.01
2021g	39.6	8	8260	Oct.	80877	28.62	<0.1	0.04	2.4	0.12	<0.1	<0.01
				Null	627081	33.69	<0.1	0.01	0.5	0.07	<0.1	<0.01
				Itv.	100112	6.70	17.9	1.12	63.9	5.25	0.6	0.18
2021h	63.8	4	10048	Oct.	97850	24.87	19.5	4.29	62.7	18.98	18.9	4.80
				Null	1533103	88.17	16.1	23.69	87.2	73.13	14.7	23.11
				Itv.	23377	0.73	0.4	0.01	2.3	0.02	0.3	<0.01
2021i	23.5	4	2559	Oct.	22982	2.90	0.4	0.02	2.4	0.03	0.3	<0.01
				Null	85782	1.79	0.2	<0.01	0.7	0.01	<0.1	<0.01
				Itv.	44604	3.70	0.1	0.02	0.0	<0.01	0.0	<0.01
2021j	15.2	20	3558	Oct.	39856	8.0	0.1	0.03	0.0	<0.01	0.0	<0.01
				Null	96125	2.49	<0.1	<0.01	0.0	<0.01	0.0	<0.01
					Batch		Incremental		Dem.-Driven		Demanded	
				Itv.	46998	3.37	3.2	0.13	15.5	0.60	0.1	0.02
average	45.9	25	4963	Oct.	46886	13.95	2.6	0.47	14.7	2.02	2.7	0.50
				Null	397800	20.85	2.6	2.97	13.8	7.43	1.6	2.31
average peak memory usage (MB)					667.6		693.1		233.1		252.4	

relative proximity of queries, edits, and their dependencies, both approaches can incur significant analysis cost, averaging around 3% and 15% respectively of that required for batch analysis.

Demanded summarization, which combines aspects of incremental and demand-driven analysis, consistently requires only <1% as much analysis work as the batch baseline, although there are some outliers where the relative locations of the program edits and demand queries require performing a larger fraction of the analysis work. Notably, the edit and fix locations in one artifact (2021h) cause the demanded configuration to perform comparably to the non-demand-driven incremental configuration, since the bug fix location depends on a large fraction of the program under analysis.

The peak memory usage of both non-demand-driven configurations is significantly higher than both demand-driven configurations, since they require analyzing a smaller fraction of the program, while adding incrementality increases peak memory usage by 4-8%. This degree of overhead is approximately as expected: incrementality allows some summaries to be persisted that would otherwise be dropped, but the overhead is small when looking at a single edit. Note also that an optimized batch analysis framework would use less memory than our batch configuration; nonetheless, the memory footprint of all of these configurations is sufficiently small to fit on modern consumer hardware.

The consistent trends in symbolic metrics across these three different domains is an indication that the rough proportion of reuse and analysis cost savings can generalize when the DSG framework is instantiated with different abstract domains.

7 SUMMARY TABULATION & WEAKENING

The previous sections describe *demanded summarization*: a framework for incremental and demand-driven abstract interpretation of recursive interprocedural programs in arbitrary abstract domains.

However, the algorithm and formalism described to this point present some opportunities for optimization. This section addresses some practical concerns that will arise in the implementation of a future analysis framework based on the theory of demanded summarization. Note that the techniques formalized in the remainder of this section are not applied in the implementation of our experimental evaluation (Section 6).

First, since our approach makes extensive use of caching and memoization, it must inevitably confront memory limitations. However, there is no means to do so in the operational semantics of Section 4. We develop and formalize in Section 7.1 some techniques to gracefully handle memory pressure in DSGs by discarding intermediate analysis state and keeping input/output procedure summaries, and show that this extension preserves the metatheoretic results of Section 5.

Then, though we fix a policy of maximizing precision in the previous sections to ensure from-scratch consistency, it may be desirable in practice to weaken or merge summaries during analysis so as to maximize incremental reuse. We formalize this notion in Section 7.2 and show that while it clearly violates from-scratch consistency, it does preserve soundness and termination.

These extensions to the semantics of Section 4 correspond to well-known optimizations which are more-or-less standard practice in batch summary-based abstract interpreters (e.g., [Calcagno and Distefano 2011; Fähndrich and Logozzo 2010; Padhye and Khedker 2013]). The purpose of this section is to relate them to our formalism and show how they preserve or weaken the metatheoretic results of Section 5, thus providing tunable parameters in the design space of practical interactive analysis tools based on demanded summarization.

We also discuss some subtleties of interprocedural dependency structures highlighted by these changes.

7.1 Summary Tabulation for Memory Pressure

Given infinite memory, an incremental analysis could simply store all previously-computed results forever, keeping them on hand in case they are ever needed. This is essentially what is formalized in Section 4. DAIGs are instantiated and added to $\mathcal{D}^*$ by Q-Instantiate and, though their *contents* are dirtied in response to edits by the semantics of Fig. 8, they are never disposed of themselves. This design keeps the presentation simple, but it is infeasible in practice. This section extends the DSG formalism to explicitly account for memory-conserving operations, showing that soundness, termination, and from-scratch consistency are preserved under the extension.

We introduce a judgment form $\mathcal{G} \dashrightarrow \mathcal{G}'$ to describe transformations of analysis state that can be applied at will by an analyzer. It is possible of course to encode these operations as extensions to the query evaluation ($\mathcal{G} \vdash_\varphi \ell \Downarrow \varphi' ; \mathcal{G}'$) or edit ($\mathcal{G} \vdash_\rho n \Leftarrow s ; \mathcal{G}'$) semantics, but such an approach ties the memory-freeing operations to analysis client interactions and complicates proofs of termination (by admitting infinite derivations that alternate between computing and discarding analysis facts).

The most direct such transformation is simply to discard procedure summary DAIGs that are not depended upon by others; this is, however, more heavy-handed than desired in most cases. In particular, note that DAIGs contain cached intermediate results at the granularity of individual semantic functions, but summary application in DSGs operates at the granularity of procedures.

Thus, instead of discarding an entire DAIG, an analyzer may exploit the fact that only its entry and exit abstract states are needed to summarize calls, by caching those two states and discarding any intermediate results.

We can extend the syntax and semantics of DSGs to express and reason about this direct summarization. Formally, we first add a summary table $\mathcal{T}$ to demanded summarization graphs, leaving the other components unchanged.

$$\begin{array}{rrl} \text{summary tables} & \mathcal{T} & ::= \varepsilon \mid \mathcal{T} ; \{\varphi\}\ \rho\ \{\varphi'\} \\ \text{summary table-equipped DSGs} & \mathcal{G} & \triangleq \langle \mathcal{D}^*, \Delta, \mathcal{T} \rangle \end{array}$$

A summary table $\mathcal{T}$ is a collection of summary triples $\{\varphi\}\ \rho\ \{\varphi'\}$ not backed by any DAIG in $\mathcal{D}^*$. For the most part, the operational semantics of Figs. 7 and 8 just work with DSGs replaced by summary table-equipped analogues and tables $\mathcal{T}$ threaded through the rules accordingly.

However, some additional rules are needed to tabulate and apply summaries. As such, we reproduce the modified rules and provide the new rules in in Fig. 10, highlighting the modifications relative to Section 4 in green.

First, we add a rule Tabulate to the $\mathcal{G} \dashrightarrow \mathcal{G}'$ judgment form, allowing the analyzer to drop a fully-solved DAIG $\mathcal{D}$ and replace it by an equivalent Hoare-style summary triple at any point.

$$\textsc{Tabulate}\quad \frac{\mathcal{D}(\mathsf{exit}(\rho)) = \varphi'}{\langle \mathcal{D}^*; (\rho, \varphi) \mapsto \mathcal{D},\ \Delta,\ \mathcal{T} \rangle \dashrightarrow \langle \mathcal{D}^*,\ \Delta,\ \mathcal{T}; \{\varphi\}\ \rho\ \{\varphi'\} \rangle}$$

Note that the dependency map Δ is unchanged when we drop a DAIG $\mathcal{D}$ and tabulate the corresponding triple $\{\varphi\}\ \rho\ \{\varphi'\}$. Thus, any analysis fact that depended on $\mathcal{D}$ now depends on the summary triple, and any DAIG or summary triple that contributed to $\mathcal{D}$ also contributed to the summary triple.

Then, we add an additional inference rule S-Apply, which allows the tabulated Hoare triples in $\mathcal{T}$ to resolve summary queries and be applied at procedure call sites.

The procedure summarization judgment $\mathcal{G} \vdash \{\varphi\}\ \rho\ \{\varphi'\}\ ; \mathcal{G}'$ effectively abstracts away the two different types of cached results in our summary table-equipped DSGs: Summarize (Section 4) interprets fully-solved DAIGs as summaries, while S-Apply (Fig. 10) interprets rows of the summary table $\mathcal{T}$ as summaries.

$$\textsc{D-Summary}\quad \frac{\Delta_{\rho,\varphi} = \emptyset \qquad \Delta' = \{\, \delta \mid (\rho,\varphi) \leftharpoonup (\_,\_) = \delta \in \Delta \,\}}{\langle D^*,\ \Delta,\ \mathcal{T}; \{\varphi\}\ \rho\ \{\_\}\rangle \vdash_{\rho,\varphi} \_ \Leftarrow \_\,; \langle D^*,\ \Delta/\Delta',\ \mathcal{T}\rangle}$$

$$\textsc{Q-Instantiate}\quad \frac{(\rho^\ell, \varphi) \notin \mathrm{dom}(\mathcal{D}^*) \qquad \{\varphi\}\ \rho^\ell\ \{\_\} \notin \mathcal{T} \qquad \langle \mathcal{D}^*; (\rho^\ell,\varphi) \mapsto \mathcal{D}^{\mathsf{init}}_{\rho^\ell,\varphi}, \Delta, \mathcal{T}\rangle \vdash_\varphi \ell \Downarrow \varphi'\,; \mathcal{G}}{\langle \mathcal{D}^*, \Delta, \mathcal{T}\rangle \vdash_\varphi \ell \Downarrow \varphi'\,; \mathcal{G}}$$

$$\textsc{S-Apply}\quad \frac{\mathcal{G} = \langle \_,\_,\mathcal{T}; \{\varphi\}\ \rho\ \{\varphi'\}\rangle}{\mathcal{G} \vdash \{\varphi\}\ \rho\ \{\varphi'\}\,; \mathcal{G}}$$

$$\textsc{E-Delegate}\quad \frac{\mathcal{G}_0 = \langle \mathcal{D}^*, \Delta\rangle \qquad \{\varphi_1, \ldots, \varphi_k\} = \left\{ \varphi \;\middle|\; \begin{array}{l} (\rho,\varphi) \in \mathrm{dom}(\mathcal{D}^*) \\ \vee\ \{\varphi\}\ \rho\ \{\_\} \in \mathcal{T} \end{array} \right\} \qquad \mathcal{G}_{i-1} \vdash_{\rho,\varphi_i} n \Leftarrow s\,; \mathcal{G}_i \quad \text{for } 1 \le i \le k}{\mathcal{G}_0 \vdash_\rho n \Leftarrow s\,; \mathcal{G}_k}$$

Fig. 10. Modifications and additions to demanded summarization query and edit semantics required to handle explicit summary tables $\mathcal{T}$. Additions are highlighted in green, and all rules *not* shown here are unchanged from Section 4. As before, underscores denote un-constrained metavariables.

Next, we add a premise $\{\varphi\}\ \rho^\ell\ \{\_\} \notin \mathcal{T}$ to the Q-Instantiate rule, preventing the instantiation of a DAIG which shadows a summary already in $\mathcal{T}$. Similarly, we tweak the definition of $\{\varphi_1, \ldots, \varphi_k\}$ in the E-Delegate rule of the edit judgment, ensuring that when a procedure is edited, not only DAIGs but also summary triples over that procedure are dirtied. Both modifications consist of checking whether a summary exists in $\mathcal{T}$ in addition to the original check whether it exists in $\mathcal{D}^*$.

Finally, we add a rule D-Summary to the dirtying judgment, which is analogous to the D-DAIG rule but discards a summary triple instead of dirtying a DAIG. Note, though, that instead of dropping only those interprocedural dependencies Δ' for now-dirtied intermediate results in $\mathcal{D}$ (as in D-DAIG) we drop all backward dependencies of the discarded triple.

Applying the Tabulate transformation allows a DSG-based analyzer to selectively *coarsen* its memoized results, achieving a significantly smaller memory footprint at the cost of fine-grained incremental reuse in cases where there is an edit inside a method whose DAIG has been dropped.

Metatheory. Moreover, extending demanded summarization graphs $\mathcal{G}$ with summary tables $\mathcal{T}$ as described here does *not* affect analysis results, since summaries in $\mathcal{T}$ are created by Tabulate, applied by S-Apply, and dirtied by D-Intraproc-Summary under exactly the conditions that apply to corresponding DAIGs in $\mathcal{D}^*$.

We do not reproduce the theorems and proofs in full here, as they are largely unchanged from Section 4, but they are available in Appendix B. The key modification required is an additional condition for semantic consistency (Definition 5.1), stating that whenever a summary triple is in $\mathcal{T}$, all of the dependency edges that would be required to compute that summary from scratch are in Δ. By ensuring that summary triples in $\mathcal{T}$ are accompanied by the proper dependencies, we can guarantee that they are invalidated as needed in response to semantically-relevant edits in other procedures.

Condensing DAIGs down to input/output summaries with Tabulate provides a semantic foundation that can be used to implement memory conservation mechanisms without jeopardizing hard-won guarantees of soundness, termination, and from-scratch consistency. Similar approaches to reducing memory usage by intelligently discarding some analysis results have been shown to significantly improve performance in IFDS-based analysis frameworks [Arzt 2021].

Classic cache replacement strategies allow an interactive analysis engine to intelligently discard or condense DAIGs at set intervals or whenever memory usage crosses some threshold. For example, the least-recently or least-frequently used DAIGs can be reduced to two-state summaries, treating the DSG as an LRU/LFU cache of intermediate analysis results and reducing the overall memory footprint of the DSG without incurring any runtime cost unless/until the summary is affected by a program edit.

Beyond the practical applicability of these operations, the relative simplicity and straightforwardness of the extension indicates the generality and foundational nature of the DSG approach. For example, we believe that the DSG approach could be adapted to operate over custom relational domains rather than state domains, essentially replacing the DAIG intraprocedural analysis black-box with a different mechanism that produces such summaries, while tracking interprocedural dependencies in much the same way as we describe here.

7.2 Summary Weakening for Reuse

In order to apply a summary at a procedure call, the core DSG operational semantics laid out in Section 4 require that its precondition *exactly* matches the callsite abstract state[7]. However, an analysis implementation may wish to maximize the reuse of previously computed summaries by applying a compatible one with a weaker-than-needed precondition instead. This is analogous to the use of monotonicity constraints in Datalog-based incremental analyses to reuse facts when there is an $\sqsubseteq$-increasing change to some input [Szabó et al. 2018].

This optimization is enabled by adding the following inference rule to the summarization judgment $\mathcal{G} \vdash \{\varphi\}\ \rho\ \{\varphi'\}\ ;\mathcal{G}'$:

S-Consequence

$$\frac{\varphi \sqsubset \varphi' \qquad \mathcal{G} \vdash \{\varphi'\}\ \rho\ \{\varphi''\}\ ;\langle \mathcal{D}^*, \Delta\ , \mathcal{T}\ \rangle}{\mathcal{G} \vdash \{\varphi\ \}\ \rho\ \{\varphi''\}\ ;\langle \mathcal{D}^*, \Delta', \mathcal{T}'\rangle} \qquad \begin{array}{rl} \text{where } \Delta' = & \Delta; (\rho, \varphi) \hookleftarrow (\rho, \varphi') \\ \text{and } \mathcal{T}' = & \mathcal{T}; \{\varphi\}\ \rho\ \{\varphi''\} \end{array}$$

Ignoring the demanded summarization graphs (i.e. everything before a turnstile or after a semicolon), this rule is precisely the familiar Hoare logic rule of consequence for preconditions [Hoare 1969]. Although it would be sound to do so, we don't include the dual postcondition rule (nor combine the two into one rule) as there is no benefit to applying it in our framework — it would simply produce less-precise analysis results.

The primed extensions of Δ and $\mathcal{T}$ in the conclusion of S-Consequence serve to avoid a subtle issue: without them, when this rule is used to weaken a summary, the Q-Apply-Summary rule adds a dependency edge corresponding to the weakened (conclusion) summary rather than the stronger (premise) summary from which it was derived. As a result, when the premise summary is dirtied the analysis would unsoundly fail to propagate that change to the callsite where its weakened form was applied. This shortcoming is fully addressed, though, by adding the derived triple to $\mathcal{T}$, along with a dependency edge in Δ on the (weaker) premise triple, since dirtying will transitively propagate the change through the derived triple $\{\varphi\}\ \rho\ \{\varphi''\}$ in $\mathcal{T}'$.[8] This does require that φ and φ be strictly ordered, as Δ' would contain spurious self-loops if they were equal. This is not a significant restriction, since "weakening" a summary to itself has no real benefit.

Weakening summaries by applying S-Consequence preserves the *soundness* of analysis results by the same reasoning that it is sound to apply the consequence rule in a standard Hoare logic. That is,

[7] modulo widening, in the case of recursive calls

[8] The issue can also be addressed without summary tables $\mathcal{T}$ by folding S-Consequence directly into Q-Apply-Summary and adjusting its dependency map accordingly. We present it this way for the sake of clarity, keeping some separation of concerns between judgment forms.

the weaker triple $\{\varphi'\}\ \rho\ \{\varphi''\}$ in its premise implies the stronger triple inferred as its conclusion: since $\varphi \sqsubset \varphi'$, any concrete state modelling φ also models φ' and is thus guaranteed to be mapped by the semantics of ρ to a state modelling φ''.

However, introducing the S-Consequence rule comes at the cost of from-scratch consistency for demanded analysis. To see why, consider an edit immediately before a procedure call that has been summarized with triple $\{\varphi_{\text{pre}}\}\ \rho\ \{\varphi_{\text{post}}\}$, and suppose that the edit results in a stronger precondition φ'_{pre} for the call. It is sound to reuse the previously computed summary via S-Consequence, deriving the same postcondition φ_{post} for the call as before. However, it is possible that a from-scratch recomputation with the newly strengthened precondition may have produced a stronger summary $\{\varphi'_{\text{pre}}\}\ \rho\ \{\varphi'_{\text{post}}\}$, so from-scratch consistency has been violated.

Thus, weakening of summaries represents a tradeoff and a tunable parameter for the design of practical analysis tools based on demanded summarization. In some circumstances, the benefit (i.e. computational savings due to increased summary reuse) may be well worth the cost of weaker precision guarantees; in others, the predictable behavior and maximal precision of a from-scratch consistent demanded analysis are more important.

8 RELATED WORK

Compositional Analysis. Sharir and Pnueli [1981] defined the *functional*, compositional approach to interprocedural analysis, and much work has followed in its footsteps. Previous approaches have achieved compositionality by tabulating function summaries [Naeem et al. 2010; Padhye and Khedker 2013; Reps et al. 1995; Sagiv et al. 1996] or by deriving two-state relational summaries [Calcagno et al. 2011; Chatterjee et al. 1999; Cousot and Cousot 2002; Dillig et al. 2011; Jeannet et al. 2010; Madhavan et al. 2015; Montagu and Jensen 2020; Salcianu 2006; Yorsh et al. 2008]. Of these, the batch analysis formulation underlying our technique is most similar to that of Padhye and Khedker [2013]. Their work tabulates function summaries keyed on a "value context," consisting of a procedure name and entrypoint abstract state, similar to our formulation. Further, the worklist algorithm described in that paper also tracks dependencies between such contexts, similar to the Δ component of DSGs (Section 4), to propagate interprocedural data flow from procedure exits to caller return sites. To perform demanded analysis and ensure from-scratch consistency, our approach rigorously defines invalidation and recomputation of such dependencies, and also exerts more fine-grained control over iteration ordering during analysis.

Monadically-parameterized semantics can also be used to define reusable abstract interpreter metatheory, offering compositionality of soundness proofs and generality over a family of different interprocedural analyses [Darais et al. 2017; Keidel and Erdweg 2019; Sergey et al. 2013]. Darais et al. [2017] cache analysis results during batch evaluation of a monadic interpreter, but rely on a finite abstract domain to ensure termination of the fixed-point computation and do not address incremental or demand-driven evalutation of their abstract interpreter.

Deductive verification systems [Barnett et al. 2011; Filliâtre and Marché 2007; Jacobs et al. 2011; Leino 2010; Müller et al. 2016] have long used compositionality to provide a real-time verification experience [Leino and Wüstholz 2015], generally with user-supplied pre- and post-conditions. While we have focused on tabulation in this paper to directly support common one-state abstractions, the two-state relational approach is conceptually the same to support for demanded analysis, except with DAIG reference cells storing two-state relations instead of single-state abstractions.

Our use of "operational" and "denotational" rather than the traditional "context-sensitive" and "functional" for Sharir and Pnueli [1981]'s two approaches to interprocedural analysis is borrowed from Jeannet et al. [2010], whose lucid treatment of the topic makes a convincing case for these more generalized terms.

Recent static analyses have achieved large scalability gains on CI servers through compositionality [Blackshear et al. 2018; Calcagno and Distefano 2011; Distefano et al. 2019; Fähndrich and Logozzo 2010]. This compositionality suggests an incremental deployment model in which procedures are reanalyzed only when they are affected by an edit [Calcagno et al. 2011].

However, naive approaches to such incrementality have yielded unreliable and/or unsound results, so existing industrial deployments of these compositional analyses have largely eschewed incremental reuse of summaries in practice. This work has directly enabled the development of rigorous incremental infrastructure for such analyses, though: a variant of demanded summarization maps is now implemented in the Infer static analyzer [Calcagno and Distefano 2011]. These developments have produced significant analysis speedups in continuous integration (on the order of 3x across all analyses, and up to 10x at the 95th percentile for certain workloads) as well as a near-total elimination of the unreliability and flakiness that characterized earlier iterations of incremental analysis in Infer [Stein 2023].

Incremental Analysis. Incremental variants of many standard compiler analyses have been studied and developed in order to support responsive continuous compilation and structured editors/integrated development environments. These include dataflow analyses [Carroll and Ryder 1988; Pollock and Soffa 1989; Ryder 1983; Zadeck 1984], pointer analyses [Gupta et al. 1993; Liu and Huang 2022; Lu et al. 2013], and attribute grammars [Demers et al. 1981; Reps 1982; Reps et al. 1983; Söderberg and Hedin 2012], which combine parse trees with semantic information and can encode many analyses including dataflow.

Recent work has also contributed incremental versions of several broader classes of program analysis, including IFDS/IDE dataflow analyses [Arzt and Bodden 2014; Do et al. 2017] and analysis DSLs based on extensions to Datalog [Szabó et al. 2018, 2021, 2016] These specialized approaches offer very effective solutions for these particular classes of program analysis, but place restrictions on abstract domains that rule out arbitrary abstract interpretations in infinite-height domains, for example by requiring domains to fall in the IFDS/IDE subset or by requiring that all domain operations are monotonic.

Some recent work by Van der Plas et al. [2020, 2023] and Garcia-Contreras et al. [2021] has explored similar approaches to incrementalization of compositional dataflow analyses, tracking and reifying inter-procedural dependencies and supporting infinite-height abstract domains. Notably, Van der Plas et al. [2023] interleave invalidation with (eager) recomputation, potentially saving a great deal of recomputation over our approach when an edit does not affect a procedure's semantics, as alluded to in Section 4.2, and Garcia-Contreras et al. [2021] also offer a demand-driven interface, analyzing only the dependencies of a given "goal" in constrained Horn clause programs. The main benefit of our approach over these closely-related works is its from-scratch consistency, offering analysis designers/implementers strong precision guarantees and simplifying debugging and deployment as a result.

These approaches are automatic (in that they require no user-provided specifications or loop invariants) and sound, but make no guarantee of from-scratch consistency and in fact violate it in some cases in order to maximize incremental reuse. On the other hand, Leino and Wüstholz [2015] propose a fine-grained incremental verification technique for the Boogie language, which verifies user-provided specifications of imperative procedures. Since these specifications include loop invariants, their algorithm can avoid altogether the issues and complexities introduced by cyclic dependencies.

Demand-Driven Analysis. Demand-driven techniques for classical dataflow analysis are similarly well-studied. The intra-procedural problem was studied by Babich and Jazayeri [1978]. Several extensions to inter-procedural analysis have been presented, for example by Reps [1994], Duesterwald et al. [1995], Horwitz et al. [1995], and Sagiv et al. [1996], and applied to problems such as array

bounds check elimination, parallel communication optimization, and integration testing [Bodík et al. 2000; Duesterwald et al. 1996; Yuan et al. 1997].

As with the incremental variants discussed in the previous section, these works are focused on finite domains (or, in the case of Sagiv et al. [1996], infinite domains of finite height).

Other types of static analysis (i.e. neither dataflow analysis nor abstract interpretation) can also be performed on-demand. Any analysis expressible as a context-free language reachability (CFL-reachability) problem can be computed in a demand-driven fashion as a "single-source" problem [Reps 1998]. As such, several papers have presented demand-driven algorithms for flow-insensitive pointer analysis [Heintze and Tardieu 2001; Späth et al. 2016; Sridharan et al. 2005]. Reference attribute grammars (RAGs) are declarative specifications of properties over ASTs (including potentially-cyclic flow analyses) which can be evaluated incrementally and on-demand [Magnusson and Hedin 2007; Söderberg and Hedin 2012]. Termination of RAG evaluation requires that all cyclic computations converge to a fixed-point in finitely-many iterations [Farrow 1986; Magnusson and Hedin 2007]; this convergence property holds for finite domains with monotone operators but may also be achieved through other means (e.g. widening). Unlike RAG-based approaches to dataflow analysis, our approach comes with proofs of termination and from-scratch consistency, and specifies the exact conditions required to ensure termination in infinite-height domains with non-monotone widening operators.

Incremental computation. Our technique is heavily influenced by dependency graph-based techniques for general incremental and self-adjusting computation [Acar et al. 2008, 2002; Demers et al. 1981; Hammer et al. 2009; Reps 1982]. Some recent work in this area has also explored dynamic dependency tracking similar to our own, but applied to build systems rather than static analysis [Konat et al. 2018].

In particular, we draw on insights from the demanded computation graphs of Adapton [Hammer et al. 2015, 2014], which extends traditional graph-based incremental computation techniques to support interactive demand-driven computations and provide robust formal guarantees.

By reifying the partial order of computation dependencies in a *demanded computation graph* (DCG), this line of work provides a powerful and general approach to the design and implementation of efficient interactive systems. However, its low-level primitives make it difficult to express the complex fixed-point computations over cyclic control-flow graphs and recursive call structures that are found in arbitrary abstract interpretations [Stein et al. 2021a]. This work takes a great deal of inspiration from demanded computation graphs, but specializes the language of demanded computations to abstract interpretation, both with syntactic structures and with query and edit semantics that dynamically modify the dependency graph to model program analysis computation.

9 CONCLUSION

We have described a novel framework for interactive abstract interpretation that simultaneously supports *demand-driven* queries, *incremental* handling of program edits, and *compositional* application of procedure summaries — all in the context of supporting arbitrary complex abstract domains with non-monotonic widening operators and recursive procedures.

The key innovation in our framework is *demanded summarization*, which instantiates demanded abstract interpretation graphs (DAIGs) on demand to synthesize summaries as needed for a compositional interprocedural analysis, where a significant technical challenge is soundly handling self-referential summaries that naturally arise from demanded summarization of recursive procedures. Our evaluation provides evidence of demanded summarization's scalability and responsiveness using synthetic benchmarks, and of its generalizability using real-world edits drawn from open-source Java programs. Together, these experiments provide evidence of the feasibility of this

approach as an interactive interface to compositional summary-based analyses with arbitrarily complex abstract domains.

ACKNOWLEDGMENTS

We thank David Flores for his valuable contributions to the preparation of the experimental evaluation in this paper. We also thank Matthew Hammer, members of the CUPLV lab, and anonymous reviewers from TOPLAS and other venues whose comments helped us improve this paper.

This research was supported in part by the National Science Foundation under grants CCF-2007024, CCF-2008369, CCF-2223825, and CCF-2223826.

REFERENCES

Umut A. Acar, Amal Ahmed, and Matthias Blume. 2008. Imperative Self-Adjusting Computation. In *Principles of Programming Languages (POPL)*. https://doi.org/10.1145/1328438.1328476

Umut A. Acar, Guy E. Blelloch, and Robert Harper. 2002. Adaptive Functional Programming. In *Principles of Programming Languages (POPL)*. https://doi.org/10.1145/1186634

Steven Arzt. 2021. Sustainable Solving: Reducing The Memory Footprint of IFDS-Based Data Flow Analyses Using Intelligent Garbage Collection. In *International Conference on Software Engineering (ICSE)*.

Steven Arzt and Eric Bodden. 2014. Reviser: Efficiently Updating IDE-/IFDS-based Data-Flow Analyses in Response to Incremental Program Changes. In *International Conference on Software Engineering (ICSE)*. https://doi.org/10.1145/2568225.2568243

Wayne A. Babich and Mehdi Jazayeri. 1978. The Method of Attributes for Data Flow Analysis: Part II. Demand analysis. *Acta Informatica* 3 (1978). https://doi.org/10.1007/BF00264320

Mike Barnett, Manuel Fähndrich, K. Rustan M. Leino, Peter Müller, Wolfram Schulte, and Herman Venter. 2011. Specification and Verification: the Spec# Experience. *Commun. ACM* 6 (2011). https://doi.org/10.1145/1953122.1953145

Sam Blackshear, Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. 2018. RacerD: Compositional Static Race Detection. *Proc. ACM Program. Lang.* OOPSLA, Article 144 (2018), 28 pages. https://doi.org/10.1145/3276514

Rastislav Bodík, Rajiv Gupta, and Vivek Sarkar. 2000. ABCD: Eliminating Array Bounds Checks on Demand. In *Programming Language Design and Implementation (PLDI)*.

Max Brunsfeld. 2021. *tree-sitter/tree-sitter: v0.20.0*. https://doi.org/10.5281/zenodo.5044536

Cristiano Calcagno and Dino Distefano. 2011. Infer: An Automatic Program Verifier for Memory Safety of C Programs. In *NASA Formal Methods (NFM)*. https://doi.org/10.1007/978-3-642-20398-5_33

Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. 2011. Compositional Shape Analysis by Means of Bi-Abduction. *J. ACM* 6, Article 26 (2011), 66 pages. https://doi.org/10.1145/2049697.2049700

Martin D. Carroll and Barbara G. Ryder. 1988. Incremental Data Flow Analysis via Dominator and Attribute Updates. In *Principles of Programming Languages (POPL)*.

Ramkrishna Chatterjee, Barbara G. Ryder, and William Landi. 1999. Relevant Context Inference. In *Principles of Programming Languages (POPL)*.

Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Principles of Programming Languages (POPL)*. https://doi.org/10.1145/512950.512973

Patrick Cousot and Radhia Cousot. 2002. Modular Static Program Analysis. In *Compiler Construction (CC)*. https://doi.org/10.1007/3-540-45937-5_13

David Darais, Nicholas Labich, Phuc C. Nguyen, and David Van Horn. 2017. Abstracting Definitional Interpreters. *Proc. ACM Program. Lang.* ICFP (2017). https://doi.org/10.1145/3110256

Alan J. Demers, Thomas W. Reps, and Tim Teitelbaum. 1981. Incremental Evaluation for Attribute Grammars with Application to Syntax-Directed Editors. In *Principles of Programming Languages (POPL)*. https://doi.org/10.1145/567532.567544

Isil Dillig, Thomas Dillig, Alex Aiken, and Mooly Sagiv. 2011. Precise and Compact Modular Procedure Summaries for Heap Manipulating Programs. In *Programming Language Design and Implementation (PLDI)*.

Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. 2019. Scaling Static Analyses at Facebook. *Commun. ACM* 8 (2019). https://doi.org/10.1145/3338112

Lisa Nguyen Quang Do, Karim Ali, Benjamin Livshits, Eric Bodden, Justin Smith, and Emerson R. Murphy-Hill. 2017. Just-in-time Static Analysis. In *Software Testing and Analysis (ISSTA)*. https://doi.org/10.1145/3092703.3092705

Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. 1995. Demand-Driven Computation of Interprocedural Data Flow. In *Principles of Programming Languages (POPL)*. https://doi.org/10.1145/199448.199461

Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. 1996. A Demand-Driven Analyzer for Data Flow Testing at the Integration Level. In *International Conference on Software Engineering (ICSE)*.

Julian Erhard, Simmo Saan, Sarah Tilscher, Michael Schwarz, Karoliine Holter, Vesal Vojdani, and Helmut Seidl. 2022. Interactive Abstract Interpretation: Reanalyzing Whole Programs for Cheap. *CoRR* (2022).

Manuel Fähndrich and Francesco Logozzo. 2010. Static Contract Checking with Abstract Interpretation. In *Formal Verification of Object-Oriented Software (FoVeOOS)*. https://doi.org/10.1007/978-3-642-18070-5_2

Rodney Farrow. 1986. Automatic Generation of Fixed-point-finding Evaluators for Circular, but Well-defined, Attribute Grammars. In *Compiler Construction (CC)*. https://doi.org/10.1145/12276.13320

Jean-Christophe Filliâtre and Claude Marché. 2007. The Why/Krakatoa/Caduceus Platform for Deductive Program Verification. In *Computer-Aided Verification (CAV)*. https://doi.org/10.1007/978-3-540-73368-3_21

Isabel Garcia-Contreras, José F. Morales, and Manuel V. Hermenegildo. 2021. Incremental and Modular Context-sensitive Analysis. *Theory Pract. Log. Program.* 2 (2021).

Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. 1993. Maintaining Views Incrementally. In *SIGMOD Conference*.

Matthew A. Hammer, Umut A. Acar, and Yan Chen. 2009. CEAL: a C-based Language for Self-Adjusting Computation. In *Programming Language Design and Implementation (PLDI)*.

Matthew A. Hammer, Jana Dunfield, Kyle Headley, Nicholas Labich, Jeffrey S. Foster, Michael W. Hicks, and David Van Horn. 2015. Incremental Computation with Names. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. https://doi.org/10.1145/2814270.2814305

Matthew A. Hammer, Yit Phang Khoo, Michael Hicks, and Jeffrey S. Foster. 2014. Adapton: Composable, Demand-Driven Incremental Computation. In *Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/2594291.2594324

Nevin Heintze and Olivier Tardieu. 2001. Demand-Driven Pointer Analysis. In *Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/378795.378802

C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 10 (1969).

Susan Horwitz, Thomas W. Reps, and Shmuel Sagiv. 1995. Demand Interprocedural Dataflow Analysis. In *Foundations of Software Engineering (FSE)*. https://doi.org/10.1145/222124.222146

Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *NASA Formal Methods (NFM)*. https://doi.org/10.1007/978-3-642-20398-5_4

Bertrand Jeannet, Alexey Loginov, Thomas W. Reps, and Mooly Sagiv. 2010. A Relational Approach to Interprocedural Shape Analysis. *ACM Trans. Program. Lang. Syst.* 2 (2010). https://doi.org/10.1145/1667048.1667050

Bertrand Jeannet and Antoine Miné. 2009. Apron: A Library of Numerical Abstract Domains for Static Analysis. In *Computer-Aided Verification (CAV)*. https://doi.org/10.1007/978-3-642-02658-4_52

Sven Keidel and Sebastian Erdweg. 2019. Sound and Reusable Components for Abstract Interpretation. *Proc. ACM Program. Lang.* OOPSLA (2019). https://doi.org/10.1145/3360602

Gabriël Konat, Sebastian Erdweg, and Eelco Visser. 2018. Scalable Incremental Building with Dynamic Task Dependencies. In *Automated Software Engineering (ASE)*.

K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*. https://doi.org/10.1007/978-3-642-17511-4_20

K. Rustan M. Leino and Valentin Wüstholz. 2015. Fine-Grained Caching of Verification Results. In *Computer-Aided Verification (CAV)*. https://doi.org/10.1007/978-3-319-21690-4_22

Bozhen Liu and Jeff Huang. 2022. SHARP: Fast Incremental Context-Sensitive Pointer Analysis for Java. *Proc. ACM Program. Lang.* OOPSLA1 (2022).

Yi Lu, Lei Shang, Xinwei Xie, and Jingling Xue. 2013. An Incremental Points-to Analysis with CFL-Reachability. In *Compiler Construction (CC)*.

Ravichandhran Madhavan, G. Ramalingam, and Kapil Vaswani. 2015. A Framework For Efficient Modular Heap Analysis. *Found. Trends Program. Lang.* 4 (2015).

Eva Magnusson and Görel Hedin. 2007. Circular Reference Attributed Grammars - their evaluation and applications. *Sci. Comput. Program.* 1 (2007). https://doi.org/10.1016/j.scico.2005.06.005

BugSwarm Maintainers. 2021a. *davidmoten-rxjava-jdbc-172208959*. https://bugswarm.org/dataset.

BugSwarm Maintainers. 2021b. *raphw-byte-buddy-234970609*. https://bugswarm.org/dataset.

BugSwarm Maintainers. 2021c. *SpigotMC-BungeeCord-130330788*. https://bugswarm.org/dataset.

BugSwarm Maintainers. 2021d. *square-okhttp-95014919*. https://bugswarm.org/dataset.

BugSwarm Maintainers. 2021e. *tananaev-traccar-164537301*. https://bugswarm.org/dataset.

BugSwarm Maintainers. 2021f. *tananaev-traccar-188473749*. https://bugswarm.org/dataset.

BugSwarm Maintainers. 2021g. *tananaev-traccar-191125671*. https://bugswarm.org/dataset.

BugSwarm Maintainers. 2021h. *tananaev-traccar-255051211*. https://bugswarm.org/dataset.
BugSwarm Maintainers. 2021i. *tananaev-traccar-64783123*. https://bugswarm.org/dataset.
BugSwarm Maintainers. 2021j. *vkostyukov-la4j-45524419*. https://bugswarm.org/dataset.
Benoît Montagu and Thomas P. Jensen. 2020. Stable Relations and Abstract Interpretation of Higher-order Programs. *Proc. ACM Program. Lang.* ICFP (2020). https://doi.org/10.1145/3409001
Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*. https://doi.org/10.1007/978-3-662-49122-5_2
Nomair A. Naeem, Ondrej Lhoták, and Jonathan Rodriguez. 2010. Practical Extensions to the IFDS Algorithm. In *Compiler Construction (CC)*. https://doi.org/10.1007/978-3-642-11970-5_8
Rohan Padhye and Uday P. Khedker. 2013. Interprocedural Data Flow Analysis in Soot using Value Contexts. In *State of the Art in Program Analysis (SOAP)*.
Lori L. Pollock and Mary Lou Soffa. 1989. An Incremental Version of Iterative Data Flow Analysis. *IEEE Trans. Software Eng.* 12 (1989).
Thomas Reps. 1998. Program Analysis via Graph Reachability. *Information and Software Technology* 11-12 (1998). https://doi.org/10.1016/S0950-5849(98)00093-7
Thomas W. Reps. 1982. Optimal-Time Incremental Semantic Analysis for Syntax-Directed Editors. In *Principles of Programming Languages (POPL)*. https://doi.org/10.1145/582153.582172
Thomas W. Reps. 1994. Solving Demand Versions of Interprocedural Analysis Problems. In *Compiler Construction (CC)*. https://doi.org/10.1007/3-540-57877-3_26
Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Principles of Programming Languages (POPL)*.
Thomas W. Reps, Tim Teitelbaum, and Alan J. Demers. 1983. Incremental Context-Dependent Analysis for Language-Based Editors. *ACM Trans. Program. Lang. Syst.* 3 (1983).
Barbara G. Ryder. 1983. Incremental Data Flow Analysis. In *Principles of Programming Languages (POPL)*. https://doi.org/10.1145/567067.567084
Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. 2018. Lessons from Building Static Analysis Tools at Google. *Commun. ACM* 4 (2018). https://doi.org/10.1145/3188720
Shmuel Sagiv, Thomas W. Reps, and Susan Horwitz. 1996. Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation. *Theor. Comput. Sci.* 1&2 (1996). https://doi.org/10.1016/0304-3975(96)00072-2
Alexandru Salcianu. 2006. *Pointer Analysis for Java Programs: Novel Techniques and Applications*. Ph.D. Dissertation. Massachusetts Institute of Technology, Cambridge, MA, USA.
Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. 2021. Lossless, Persisted Summarization of Static Callgraph, Points-To and Data-Flow Analysis. In *Object-Oriented Programming (ECOOP)*.
Ilya Sergey, Dominique Devriese, Matthew Might, Jan Midtgaard, David Darais, Dave Clarke, and Frank Piessens. 2013. Monadic Abstract Interpreters. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. https://doi.org/10.1145/2491956.2491979
Micha Sharir and Amir Pnueli. 1981. Two Approaches to Interprocedural Data Flow Analysis. In *Program Flow Analysis: Theory and Applications*.
Emma Söderberg and Görel Hedin. 2012. Incremental Evaluation of Reference Attribute Grammars using Dynamic Dependency Tracking. LU-CS-TR:2012-249 (2012).
Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. 2016. Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java. In *Object-Oriented Programming (ECOOP)*. https://doi.org/10.4230/DARTS.2.1.12
Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. 2005. Demand-Driven Points-To Analysis for Java. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. https://doi.org/10.1145/1094811.1094817
Benno Stein. 2023. Incremental Analysis in Infer. In *Infer Practitioners Workshop, PLDI 2023*. https://pldi23.sigplan.org/details/infer-2023-papers/8.
Benno Stein, Bor-Yuh Evan Chang, and Manu Sridharan. 2021a. Demanded Abstract Interpretation. In *Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/3453483.3454044
Benno Stein, David Flores, Bor-Yuh Evan Chang, and Manu Sridharan. 2021b. *DAI: Demanded Abstract Interpretation*. https://github.com/cuplv/dai.
Tamás Szabó, Gábor Bergmann, Sebastian Erdweg, and Markus Voelter. 2018. Incrementalizing Lattice-based Program Analyses in Datalog. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. https://doi.org/10.1145/3276509
Tamás Szabó, Sebastian Erdweg, and Gábor Bergmann. 2021. Incremental Whole-Program Analysis in Datalog with Lattices. In *Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/3453483.3454026

Tamás Szabó, Sebastian Erdweg, and Markus Voelter. 2016. IncA: a DSL for the Definition of Incremental Program Analyses. In *Automated Software Engineering (ASE)*. https://doi.org/10.1145/2970276.2970298

David A. Tomassi, Naji Dmeiri, Yichen Wang, Antara Bhowmick, Yen-Chuan Liu, Premkumar T. Devanbu, Bogdan Vasilescu, and Cindy Rubio-González. 2019. BugSwarm: Mining and Continuously Growing a Dataset of Reproducible Failures and Fixes. In *International Conference on Software Engineering (ICSE)*. https://doi.org/10.1109/ICSE.2019.00048

Jens Van der Plas, Quentin Stiévenart, Noah Van Es, and Coen De Roover. 2020. Incremental Flow Analysis through Computational Dependency Reification. In *Source Code Analysis and Manipulation (SCAM)*.

Jens Van der Plas, Quentin Stiévenart, and Coen De Roover. 2023. Result Invalidation for Incremental Modular Analyses. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*.

WALA. 2021. *T.J Watson Libraries for Analysis (WALA)*. https://wala.sourceforge.net.

Greta Yorsh, Eran Yahav, and Satish Chandra. 2008. Generating Precise and Concise Procedure Summaries. In *Principles of Programming Languages (POPL)*. https://doi.org/10.1145/1328438.1328467

Xin Yuan, Rajiv Gupta, and Rami G. Melhem. 1997. Demand-Driven Data Flow Analysis for Communication Optimization. *Parallel Process. Lett.* 4 (1997).

F. Kenneth Zadeck. 1984. Incremental Data Flow Analysis in a Structured Program Editor. In *Compiler Construction (CC)*. https://doi.org/10.1145/502874.502888

A PROOFS FOR SECTION 5 (FROM-SCRATCH CONSISTENCY)

Definition A.1 (DSG Semantic Consistency). We say that a DSG $\mathcal{G} = \langle \mathcal{D}^*, \Delta \rangle$ is *semantically consistent* when it is syntactically well-formed and consistent with the program structure and underlying abstract interpretation semantics.

- Each constituent DAIG $\mathcal{D}^*(\rho, \varphi)$ is well-formed and consistent with the corresponding procedure CFG $P(\rho)$ and intraprocedural abstract semantics $\langle \Sigma^\sharp, \varphi, [\![\cdot]\!]^\sharp, \sqsubseteq, \sqcup, \nabla \rangle$.
- Each return-site abstract state has a corresponding dependency edge from its callee in Δ. That is, if n names a non-empty ref cell in some $\mathcal{D} = \mathcal{D}^*(\rho, \varphi)$ with a ρ'-labeled edge to n from n', then either
  - $(\rho, \varphi) \overset{n}{\hookleftarrow} (\rho', \mathcal{D}(n')) \in \Delta$ (when $\rho \neq \rho'$) or
  - $(\rho, \varphi) \overset{n}{\hookleftarrow} (\rho', \varphi \nabla \mathcal{D}(n')) \in \Delta$ (when $\rho = \rho'$).
- Dependency edges $(\rho', \varphi') \overset{n}{\hookleftarrow} (\rho, \varphi)$ in Δ are consistent with the relevant DAIGs, in that
  - $\mathcal{D}^*(\rho', \varphi')(n) = \mathcal{D}^*(\rho, \varphi)(\mathsf{exit}(\rho))$ (when $\rho \neq \rho'$) or
  - $\mathcal{D}^*(\rho', \varphi')(n) \sqsupseteq \mathcal{D}^*(\rho, \varphi)(\mathsf{exit}(\rho))$ (when $\rho = \rho'$).
- Analysis results in DAIGs are equal to the corresponding invariants produced by batch tabulation, where the domains coincide: for all $\mathcal{D}^*(\rho, \varphi)(\underline{\ell})$ where $(\varphi, \ell) \in \mathsf{dom}(I)$, $\mathcal{D}^{\mathcal{G}}_{\rho,\varphi}(\underline{\ell}) = I(\varphi, \ell)$.

Lemma A.1 (Initial DSG Consistency). *The DSG $\langle \varepsilon, \varepsilon \rangle$ with no cached results is semantically consistent.*

Proof. All four conditions of Definition 5.1 are universally quantified over $\mathcal{D}$ or Δ, which are empty here and thus vacuously satisfied. □

Lemma A.2 (Consistency Preservation). *If $\mathcal{G}$ is semantically consistent (with respect to a program P) then:*
if $\mathcal{G} \vdash_\varphi \ell \Downarrow \varphi' ; \mathcal{G}'$ then $\mathcal{G}'$ is semantically consistent, and
if $\mathcal{G} \vdash_\rho n \Leftarrow s ; \mathcal{G}'$ then $\mathcal{G}'$ is semantically consistent (with respect to the edited version of P).

Proof. We will show that each of the the four conditions of Definition 5.1 is preserved in turn.

- *Each $\mathcal{D} \in \mathcal{D}^*$ is well-formed and consistent.* This follows directly from [], since the only way we instantiate and modify DAIGs is through the inference rules defined there. (modulo ρ-labeled edges, over which DAIG well-formedness makes no claims, but we will handle in the third bullet point)
- *Each return-site abstract state has in Δ a corresponding dependency edge from its callee.* The only query rules that modify return site abstract states either add the requisite edge to Δ (SQ-Other-Proc, SQ-Other-Pre, SQ-Self) or have a premise that guarantees the requisite edge is in Δ (F-Step, via $\mathsf{R}_{\rho,\varphi}$)
  The dirtying rule D-DemandedSummaries empties a return-site n' and drops the corresponding edge from Δ, while D-DAIG throws away any dangling dependency edges Δ' after dirtying intraprocedurally.
- *Each edge in Δ is consistent with the values on either side.* First, note that by the same argument as the previous case, dependency edges are always removed when the return site they point to is dirtied. Then, we need only consider the SQ-* rules which add dependency edges to Δ: SQ-Other-Proc, SQ-Other-Pre, and SQ-Self. The dependency edges added in SQ-Other-* both satisfy the condition, since the value φ_{post} at the callee exit is written directly to the return site n.
  The edge added in SQ-Self temporarily violates the condition, but the conclusion of its final premise guarantees that the resulting $\mathcal{G}'$ satisfies it (by the $\mathsf{R}_{\rho,\varphi}(\mathcal{G}')$ premise of F-Converge).
- *Cached results agree with batch tabulation.*

To show that this property is preserved under edits, first note that E-Delegate (the only edit rule) applies the edit by dirtying each constituent DAIG over the edited procedure. So, we will proceed by structural induction on the derivation of the i-indexed dirtying premise of that rule, showing that all possibly-affected analysis results are dirtied:

- Case D-DAIG: Since there are no dependency edges on this procedure in Δ, the only cached analysis results affected by the edit are in the DAIG $\mathcal{D}^*(\rho, \varphi)$. By [], the local dirtying in that DAIG is sound.
- Case D-DemandedSummaries: Because the dependency edge $(\rho', \varphi') \overset{n'}{\hookleftarrow} (\rho, \varphi)$ is in Δ, the analysis result at n' in $\mathcal{D}^*(\rho', \varphi')$ relied on this summary. By the inductive hypothesis and preservation of the second condition of semantic consistency, the first premise dirties all results that depended transitively on that result, producing a $\mathcal{G}$ whose dependency map reflects all other uses of this summary. By the inductive hypothesis, the second premise also preserves cache agreement with batch tabulation, so the final resulting $\mathcal{G}'$ contains only those results that did not depend on the edit.

In order to show that it is preserved under queries, we proceed by structural induction on the derivation of $\mathcal{G} \vdash_\varphi \ell \Downarrow \varphi' ; \mathcal{G}'$:

- Case Q-Instantiate: The initial DSG of the premise is consistent because the initial DAIG has only one non-empty cell (its entry) where $\mathcal{D}^{\mathsf{init}}_{\rho^\ell, \varphi}(\mathsf{entry}(\rho^\ell)) = \varphi = I(\varphi, \mathsf{entry}(\rho^\ell))$.
- Cast Q-Delegate: By from-scratch consistency of DAIGs.
- Case Q-Apply-Summary: First, since the only inference rule of the $\mathcal{G} \vdash \{\varphi\}\, \rho\, \{\varphi'\} ; \mathcal{G}'$ judgment is Summarize, we apply the inductive hypothesis at its premise to get that $\mathcal{G}'$ is consistent and thus $\varphi_{\mathsf{post}} = I(\varphi_{\mathsf{pre}}, \mathsf{exit}(\rho))$. Then, writing φ_{post} to the return site n is exactly summary application in tabulation, so the premise of the inductive premise is consistent and therefore so is its conclusion by the inductive hypothesis.

□

Theorem A.2 (Termination). *Queries and edits terminate:*

- *For all φ, $\ell \in L$, and consistent $\mathcal{G}$, there exist φ' and $\mathcal{G}'$ such that $\mathcal{G} \vdash_\varphi \ell \Downarrow \varphi' ; \mathcal{G}'$.*
- *For all s, n, ρ, and consistent $\mathcal{G}$ where n names a CFG edge in E^ρ, there exists a $\mathcal{G}'$ such that $\mathcal{G} \vdash_\rho n \Leftarrow s ; \mathcal{G}'$.*

We show each bullet point separately. First, for queries:

Proof. Let $tc(\rho)$ denote the number of non-ρ transitive callees of ρ in P and note that $tc(\rho') < tc(\rho)$ whenever ρ calls ρ' and $\rho \neq \rho'$. (this does not hold if we don't exclude ρ itself, since e.g. a callgraph where ρ calls ρ' and ρ' calls itself would violate it)

We'll proceed by induction on $tc(\rho^\ell)$, where ρ^ℓ is the procedure containing the query location ℓ.

Note that Q-Instantiate can apply at most once for each ρ^ℓ, φ pair and so we will ignore it throughout this proof, assuming DAIGs are materialized whenever needed.

We will split each case into two subcases based on whether or not ρ^ℓ is recursive (i.e. contains a call to itself).

- **Base case** ($tc(\rho^\ell) = 0$, non-recursive ρ^ℓ): There are no calls in ρ^ℓ, so Q-Delegate applies, its premise guaranteed by intraprocedural DAIG query termination [].
- **Base case** ($tc(\rho^\ell) = 0$, recursive ρ^ℓ): A query against the relevant sub-DAIG either returns a result (i.e. $\mathcal{D}^{\mathcal{G}}_{\rho^\ell, \varphi} \vdash \underline{\ell} \Rightarrow \varphi' ; \mathcal{D}'$) or is blocked at a recursive callsite (i.e. $\mathcal{D}^{\mathcal{G}}_{\rho^\ell, \varphi} \vdash \underline{\ell} \overset{n}{\rightsquigarrow} (\rho, \varphi) ; \mathcal{D}'$).

  (1) $\mathcal{D}^{\mathcal{G}}_{\rho^\ell, \varphi} \vdash \underline{\ell} \Rightarrow \varphi' ; \mathcal{D}'$

  The sub-DAIG query completes on its own, so Q-Delegate applies and the query terminates with its result.

(2) $\mathcal{D}^{\mathcal{G}}_{\rho^\ell,\varphi} \vdash \underline{\ell} \overset{n}{\rightsquigarrow} (\rho^\ell, \varphi')$; $\mathcal{D}'$:
Only Q-Apply-Summary applies, so we must derive a summary query.
Since $\rho^\ell = \rho^\ell$, we consider only SQ-Other-Pre and SQ-Self, depending on whether or not $\varphi' \sqsubseteq \varphi$.
- SQ-Self: We proceed through the F-* rules for fixed-point computation to derive the final premise. Due to the convergence condition of ∇, F-Step can only apply finitely many times before F-Converge applies. The $\mathcal{G} \vdash \{\varphi\}\ \rho\ \{\varphi'\}$; $\mathcal{G}'$ premises of the F-* rules similarly converge either via Q-Delegate, or finitely many applications of Q-Apply-Summary (since there are finitely many syntactic callsites in a given procedure).
- SQ-Other-Pre: By the convergence condition of ∇, only finitely many new DAIGs over ρ^ℓ may be instantiated via the $\mathcal{G} \vdash \{\varphi \nabla \varphi'\}\ \rho\ \{\varphi_{\text{post}}\}$; $\mathcal{G}'$ premise, eventually yielding one where SQ-Self applies instead of SQ-Other-Pre and terminates by the argument of the previous case, allowing any intermediate ρ DAIGs also to terminate via intraprocedural analysis with Q-Delegate after their demanded summaries return.

- **Inductive case**($tc(\rho^\ell) = n$, non-recursive ρ^ℓ): A query for $\underline{\ell}$ under precondition φ against the relevant sub-DAIG $\mathcal{D}^{\mathcal{G}}_{\rho^\ell,\varphi}$ either returns a result (in which case Q-Delegate applies and the query terminates), or is blocked at some callsite (i.e. $\mathcal{D}^{\mathcal{G}}_{\rho^\ell,\varphi} \vdash \underline{\ell} \overset{n}{\rightsquigarrow} (\rho, \varphi')$; $\mathcal{D}'$), in which case the only applicable rule is Q-Apply-Summary.
We can derive the summary premise $\mathcal{G} \vdash \{\varphi_{\text{pre}}\}\,\rho\,\{\varphi_{\text{post}}\}$; $\mathcal{G}'$ of Q-Apply-Summary via Summarize, using the inductive hypothesis to derive its premise because $tc(\rho) < n$. After writing the resulting φ_{post} to n in $\mathcal{D}'$ and updating Δ accordingly, we reissue the query for ℓ under φ (i.e. the recursive final premise of Q-Apply-Summary)
Note that the inductive hypothesis does not apply here, since the reissued query is in the same procedure ρ^ℓ. However, since ρ^ℓ is non-recursive, either Q-Delegate or Q-Apply-Summary must apply again.
If it is Q-Delegate, then the query terminates with its result. On the other hand, if it is Q-Apply-Summary, then that reissued query may itself reissue a query for which Q-Apply-Summary applies, and so on. Note, though, that each time the query is reissued it is against a DSG with one fewer empty return site abstract state reference cell in its (ρ^ℓ, φ) DAIG. Thus, since there are only finitely many return sites in ρ^ℓ, eventually Q-Delegate will apply and the query will terminate.
- **Inductive case**($tc(\rho^\ell) = n$, recursive ρ^ℓ):
This case is shown by combining the arguments of the previous two cases: the inductive hypothesis ensures termination of any sub-queries at non-recursive calls (as in the non-recursive inductive case), while widening ensures termination of any sub-queries at recursive calls and convergence of the in-place fixed-point computation performed by F-Step and F-Converge (as in the recursive base case).

□

And next, for edits:

Proof. Because $\mathcal{D}^*$ and therefore k are finite, this amounts to showing that each of the k-indexed dirtying premises terminates, i.e. that for all ρ, φ, n, s, and semantically consistent $\mathcal{G}$, there exists $\mathcal{G}'$ such that $\mathcal{G} \vdash_{\rho,\varphi} n \Leftarrow s$; $\mathcal{G}'$. We proceed by induction on the size of the dependency map Δ. In the base case when $\Delta_{\rho,\varphi}$ is empty (as is necessarily the case when $|\Delta| = 0$), D-DAIG applies — we dirty the $\mathcal{D}^*(\rho, \varphi)$ from n to produce $\mathcal{D}'$, discard any dependencies Δ' of dirtied local results, construct $\mathcal{G}' = \langle \mathcal{D}^*[\mathcal{D}'/(\rho, \varphi)], \Delta/\Delta' \rangle$, and terminate.

In the inductive case with $|\Delta| = n$ and $|\Delta_{\rho,\varphi}| > 0$, D-DemandedSummaries applies. Its first inductive premise is with $|\Delta| = n - 1$, so by the inductive hypothesis it terminates with some $\mathcal{G}$. That $\mathcal{G}$ also has $|\Delta \leq n - 1$ since the dirtying rules only remove and never add dependency edges in Δ, so it also terminates by the inductive hypothesis, and we terminate with its result $\mathcal{G}'$. □

Theorem A.3 (From-Scratch Consistency). *Query results are equal to the corresponding invariant computed by tabulation: if $\mathcal{G}$ is semantically consistent, $\mathcal{G} \vdash_\varphi \ell \Downarrow \varphi' ; \mathcal{G}'$, and $(\varphi, \ell) \in \mathsf{dom}(I)$ then $\varphi' = I(\varphi, \ell)$.*

Proof. Having shown semantic consistency preservation and initial-DSG semantic consistency, the proof of this property is straightforward. Since

(1) the fourth property of semantic consistency ensures that DAIG results are consistent with tabulation results,
(2) Q-Delegate (which reads its result directly from the relevant procedure DAIG) is the only $\mathcal{G} \vdash_\varphi \ell \Downarrow \varphi' ; \mathcal{G}'$ rule with no inductive premise, and
(3) the other two rules $\mathcal{G} \vdash_\varphi \ell \Downarrow \varphi' ; \mathcal{G}'$ return the result of their inductive premise,

all query results must correspond to DAIG cell values, which are themselves consistent with batch tabulation results where the domains coincide. □

Corollary A.4 (Soundness). *Query results are sound with respect to the concrete semantics: If $\mathcal{G} \vdash \ell \Downarrow \varphi ; \mathcal{G}'$ and $\mathcal{G}$ is semantically consistent then $\sigma \models \varphi$ for all $\sigma \in [\![\ell]\!]^*$*

Proof. Soundness follows directly from from-scratch consistency: results are equal to those of the underlying batch analysis, which is itself sound. □

B PROOFS FOR SECTION 7 (SUMMARY TABULATION & WEAKENING)

First, some modifications are required in the definition of DSG Semantic Consistency (Definition 5.1) to accommodate the summary triples in $\mathcal{T}$.

Definition B.1 (DSG Semantic Consistency with Summary Tables). Conditions (1) and (2) of Definition 5.1 are unmodified and therefore elided here. Conditions (3) and (4) are modified and a condition (5) added as follows

(3) Dependency edges $(\rho', \varphi') \overset{n}{\hookleftarrow} (\rho, \varphi)$ in Δ are consistent with the relevant DAIGs and summary triples, such that whenever $(\rho', \varphi) \in \mathsf{dom}(\mathcal{D}^*)$ we have either
   - $\mathcal{D}^*(\rho', \varphi')(n) = \varphi_{\text{callee-exit}}$ (when $\rho \neq \rho'$) or
   - $\mathcal{D}^*(\rho', \varphi')(n) \sqsupseteq \varphi_{\text{callee-exit}}$ (when $\rho = \rho'$).

   where $\varphi_{\text{callee-exit}} = \begin{cases} \varphi' & \text{if } \exists\ \{\varphi\}\ \rho\ \{\varphi'\} \in \mathcal{T} \\ \mathcal{D}^*(\rho, \varphi)(\mathsf{exit}(\rho)) & \text{otherwise} \end{cases}$
(4) Analysis results in DAIGs and summary triples are equal to the corresponding invariants produced by batch tabulation, where the domains coincide:
   - $\mathcal{D}^{\mathcal{G}}_{\rho,\varphi}(\underline{\ell}) = I(\varphi, \ell)$ for all $\mathcal{D}^*(\rho, \varphi)(\underline{\ell})$ where $(\varphi, \ell) \in \mathsf{dom}(I)$, and
   - $\varphi' = I(\varphi, \mathsf{exit}(\rho))$ for all $\{\varphi\}\ \rho\ \{\varphi'\} \in \mathcal{T}$ where $(\varphi, \mathsf{exit}(\rho)) \in \mathsf{dom}(I)$.
(5) For all $\{\varphi\}\rho\{\varphi'\} \in \mathcal{T}$, we have that $\Delta \supseteq \Delta'$ where Δ' is defined by $\langle \varepsilon, \varepsilon, \varepsilon \rangle \vdash_\varphi \mathsf{exit}(\rho) \Downarrow \varphi' ; \langle \_, \Delta', \_ \rangle$

The tweaks to (3) and (4) are straightforward, simply allowing for the fact that analysis results can now be stored either in $\mathcal{D}^*$ or $\mathcal{T}$, rather than just in $\mathcal{D}^*$. In the first bullet point, we've inserted $\varphi_{\text{callee-exit}}$ into the two equations where we previously just had $\mathcal{D}^*(\rho, \varphi)(\mathsf{exit}(\rho))$. In the second bullet point, we've added the additional condition that summary triples agree with batch tabulation, where previously it just referred to DAIG analysis results.

The newly-added condition (5) states that, whenever we have a summary triple in $\mathcal{T}$, we also have all of the dependency edges in Δ that would be required to compute that summary from scratch. By ensuring that summary triples in $\mathcal{T}$ are accompanied by the proper dependencies, we can guarantee that they are invalidated as needed in response to semantically relevant edits to other procedures.

Theorem B.2 (Consistency Preservation under Tabulate). *If $\mathcal{G} \dashrightarrow \mathcal{G}'$ and $\mathcal{G}$ is semantically consistent with respect to P, then $\mathcal{G}'$ is also semantically consistent with respect to P.*

Proof Sketch. Conditions (1) and (2) are vacuously preserved. Both modified conditions (3) and (4) are ensured by Tabulate and Definition 5.1 of $\mathcal{G}$, since the postcondition of a tabulated triple is exactly the exit abstract state of the discarded DAIG. Condition (5) is similarly guaranteed by the definition of Tabulate, where Δ contains all requisite dependencies for $\mathcal{D}$ (by semantic consistency of $\mathcal{G}$) and is unchanged in $\mathcal{G}'$. □

Theorem B.3 (Consistency Preservation). *Semantic consistency of summary table-equipped DSGs is preserved under queries and edits as in Lemma 5.1.*

Proof Sketch. The proof is largely unchanged for the original 4 conditions of semantic consistency. Condition (5) is ensured because summary triples in $\mathcal{T}$ are never introduced by queries or edits (only by the Tabulate operation) and are discarded by the D-Summary rule whenever any transitively-reachable dependency in Δ is dirtied by an edit. □

Theorem B.4 (Termination). *Queries and edits terminate in semantically consistent summary table-equipped demanded summarization graphs.*

Proof Sketch. The added rules S-Apply and D-Summary clearly terminate — S-Apply trivially, D-Summary by the same argument given for D-DAIG in Section 5. The modified rules Q-Instantiate and E-Delegate also terminate, both by the same arguments given originally in Section 5. □

Theorem B.5 (From-Scratch Consistency and Soundness). *Demanded analysis in summary table-equipped demanded summarization graphs produces from-scratch consistent (and therefore sound) results.*

Proof Sketch. The Tabulate operation only produces summaries that are identical to their DAIG analogues, which we know to be from-scratch consistent. Furthermore, E-Delegate and D-Summary ensure that any summary triple potentially affected by an edit is discarded. Therefore, summary triples are only applied under the same conditions as their DAIG analogues would have been, so the abstract semantics are identical to those of Section 4. □

REFERENCES FOR THE APPENDIX

Benno Stein, Bor-Yuh Evan Chang, and Manu Sridharan. Demanded Abstract Interpretation. In *Programming Language Design and Implementation (PLDI)*, 2021.