Demanded Abstract Interpretation

Anonymous Author(s)

Abstract

We consider the problem of making expressive static analyzers *interactive*. Formal static analysis is seeing increasingly widespread adoption as a tool for verification and bug-finding, but even with powerful cloud infrastructure it can take minutes or hours to get *batch* analysis results after a code change. While existing techniques offer some demand-driven *or* incremental aspects for certain classes of analysis, the fundamental challenge we tackle is doing *both* for arbitrary abstract interpreters.

Our technique, *demanded abstract interpretation*, lifts program syntax and analysis state to a dynamically evolving graph structure, in which program edits, client-issued queries, and evaluation of abstract semantics are all treated uniformly. The key difficulty addressed by our approach is the application of general incremental computation techniques to the complex, cyclic dependency structure induced by abstract interpretation of loops with widening operators. We prove that desirable abstract interpretation meta-properties, including soundness and termination, are preserved in our approach, and that demanded analysis results are equal to those computed by a batch abstract interpretation. Experimental results suggest promise for a prototype demanded abstract interpretation framework: by combining incremental and demand-driven techniques, our framework consistently delivers analysis results at interactive speeds, answering 95% of queries within 1.2 seconds.

1 Introduction

Static analysis is seeing increasing real-world adoption for verification and bug finding, particularly as part of continuous integration (CI) and code review processes [9, 34]. However, a pain point with these deployments is that developers cannot get quick local analysis results for code they are editing; ideally, updated results would appear nearly instantly in their IDE. In this paper, we present an *interactive* analysis engine designed to handle local queries and edits efficiently, which can complement a *batch* engine that exhaustively analyzes a fixed program, for example by quickly verifying whether a local change silences an alarm raised in CI.

The well-known techniques of demand-driven analysis and incremental analysis help address this challenge. Demand-driven analyses compute only those results needed to answer a set of extrinsically-provided queries, while incremental analyses speed up re-analysis of an edited program by re-using as many previously-computed results as possible.

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

Powerful frameworks for incremental (e.g., [5, 42]) *or* demand-driven (e.g., [23]) static analysis do exist, but nearly all such frameworks target restricted analysis domains (e.g., finite or finite-height domains), whereas well-known analyses like octagon and shape analysis require an infinite-height abstract domain. There are also approaches to adapt summary-based analyses to offer coarse-grained method- or file-level incrementality (e.g., [9, 14, 18]). Though these approaches effectively scale to industrial codebases in CI pipelines, they are not intended to achieve real-time interactivity during the development process.

In contrast, our aim is to support fine-grained incremental *and* demand-driven analysis over *arbitrary* abstract domains expressed in general-purpose languages, thus enabling the reuse of existing optimized abstract domain implementations at interactive speeds. To our best knowledge, no general technique exists to automatically compute a demand-driven or incremental version of an arbitrary abstract interpretation with arbitrary widening operators, a key requirement to ensure termination in more complex analyses.

This paper presents *demanded abstract interpretation*, an abstract interpretation framework with first-class support for *both* demand-driven and incremental analysis. We build on abstract interpretation [11], which provides a methodology for expressing static analyses and guaranteeing their correctness, and take inspiration from work on general incremental computation using "demanded computation graphs" [21].

Our framework reifies the abstract interpretation (AI) of a program as a dynamically evolving *demanded abstract interpretation graph* (DAIG), which explicitly represents program statements, abstract states, and the dependency structure of analysis computations. In this representation, program edits, client-issued queries, and the evaluation of abstract semantics can all be treated uniformly. Cyclic control flow is a key difficulty for this approach, since cyclic dependencies lead to unclear evaluation semantics. We define an operational semantics for DAIGs that preserves an acyclic invariant while modifying and extending the graph on demand, thus soundly analyzing loops with guaranteed termination, assuming termination of the underlying abstract interpretation.

The DAIG encoding and evaluation enables efficient abstract interpretation in an *interactive* mode, analyzing a minimal number of statements to respond to queries with maximal reuse of previously-computed results. In particular, this paper makes the following key contributions:

- We introduce a framework for *demanded abstract interpretation* in which program syntax and analysis computation structure are reified into *demanded abstract interpretation graphs* (DAIGs) (Section 4).

- We specify an operational semantics for DAIGs that realizes incremental updates and demand-driven evaluation via demanded unrolling of abstract interpretation fixed-point computations (Section 5).
- We prove that demanded abstract interpretation preserves soundness and termination, and that its results are from-scratch consistent with classical abstract interpretation by global fixed-point iteration (Section 6).
- We provide evidence for the expressivity and efficacy of demanded abstract interpretation using a prototype framework instantiated with interval, octagon, and shape domains and supporting context-sensitive interprocedural analysis (Section 7).

2 Overview

Fig. 1 shows a simple imperative program that appends two linked lists. Given well-formed (i.e., null-terminated and acyclic) input lists p and q, append must return a well-formed list and not dereference **null** in order to be correct. These properties can be verified using a separation logic, abstract interpretation-based shape analysis [7, 15, 26], tracking facts like $\texttt{lseg}(p, null)$ to represent well-formedness of list p. Our goal is to enable interactive performance for arbitrary abstract interpretations, including such analyses, in response to a user's edits and queries.

In this section, we illustrate our approach to *demanded abstract interpretation* by example. We demonstrate how abstract interpretation of forwards control flow is reified in a DAIG (Section 2.1), and then show how the DAIG supports both demand-driven and incremental interactions with the underlying abstract interpretation (Section 2.2). Finally, we highlight key difficulties introduced by cyclic control flow, show how analysis thereof can be encoded acyclically, and demonstrate how our operational semantics evolve the DAIG on-demand to soundly compute fixed points (Section 2.3).

2.1 Reifying Abstract Interpretation in DAIGs

The append procedure from Fig. 1 may equivalently be represented as a control-flow graph (CFG) as shown in Fig. 2, with vertices for program locations and edges labelled by atomic program statements.[1] A classical abstract interpreter analyzes such a program by starting with some initial abstract state at the entry location and applying an abstract transfer function $[\![\cdot]\!]^\sharp$ to interpret statements, a join operator $\sqcup$ at nodes with multiple predecessors, and a widen operator ∇ at cycles as needed until a fixed point is reached.

The demanded abstract interpretation graph (DAIG) shown in Fig. 3 reifies the computational structure of such an abstract interpretation of the Fig. 2 CFG. Its vertices are uniquely-named mutable reference cells containing program syntax or abstract state, and its edges fully specify the computations

[1] As is standard, we have simply broken down the guard conditions from **if** and **while** into **assume** guard statements for each side of a branch.

```
    function append(p: List, q: List): List {
ℓ0    if (p == null) {
ℓ1      return q; }
ℓ2    var r: List = p;
ℓ3    while (r.next != null) {
ℓ4      r = r.next; }
ℓ5    r.next = q;
ℓ6    return p;
ℓret }
```

Figure 1. A procedure to append two linked lists. The labels ℓ_i mark program locations in its control flow.

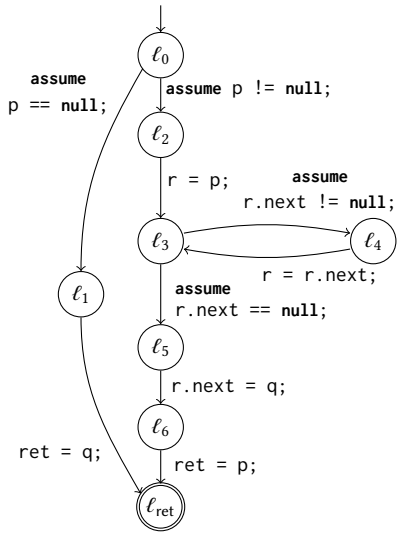


Figure 2. The control-flow graph (CFG) of the append procedure from Fig. 1.

of an abstract interpretation. Names identify values for reuse across edits and queries, and hence must *uniquely* identify the inputs and intermediate results of the abstract interpretation. In Fig. 3 and throughout this paper, *underlined* symbols denote a name derived from that symbol: hashes, essentially.

To encode abstract interpretation computations, DAIG edges are labelled by a symbol for an abstract interpretation function and connect cells storing the function inputs to the cell storing the output, capturing the dependency structure of the computation in an expected manner.[2] For example, the computation of the abstract transfer function over the CFG edge $\ell_0 \to \ell_1$ is encoded in Fig. 3 as a DAIG edge with input cells $\underline{\ell_0}$ and $\underline{\ell_0} \cdot \underline{\ell_1}$ (respectively containing the fixed-point state at ℓ_0 and the corresponding statement s_0 : **assume**(p == **null**)), labelled by the abstract transfer function symbol $[\![\cdot]\!]^\sharp$.

Note that cells containing program syntax (shaded in Fig. 3) and analysis state (unshaded in Fig. 3) are treated

[2] More precisely, DAIGs have *hyper*-edges, since they connect multiple sources (function inputs) to one destination (function output).

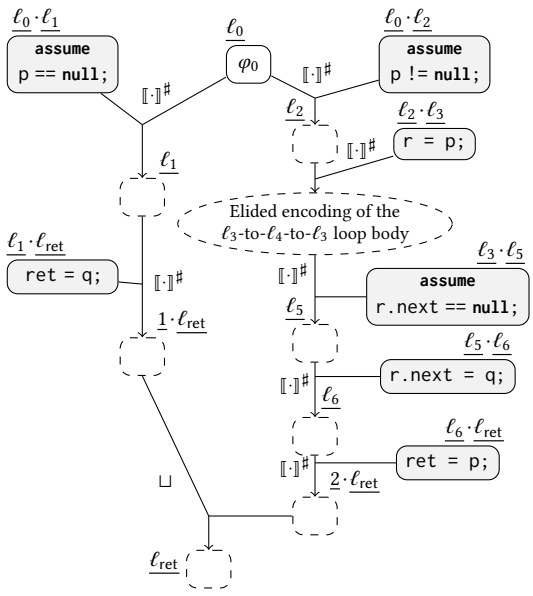


Figure 3. A demanded abstract interpretation graph (DAIG) for the program given in Fig. 1 before any queries are issued. The elided loop encoding is shown in Fig. 4c.

uniformly in the DAIG semantics; we differentiate them visually only for clarity. Statement modifications during program changes and abstract state updates during analysis computation both correspond simply to edits to the relevant mutable reference cells.

2.2 Demand-Driven and Incremental Analysis

Next, we demonstrate how a DAIG encoding naturally supports demand-driven and incremental analysis. We use the aforementioned shape-analysis domain for our example, a separation logic-based domain with a "list segment" primitive $\texttt{lseg}(x, y)$ that abstracts the heaplet containing a list segment from x to y.[3] This domain is of infinite height, absent a best abstraction function, and with complex widening operators, and therefore incompatible with previous frameworks that restrict the domain form.

In Fig. 4a, we show the result of evaluating a demand query on our example DAIG. Suppose a client issues a query for the $1 \cdot \ell_{ret}$ cell in Fig. 3, the abstract state corresponding to the `return q` statement at ℓ_1 in Fig. 1. Since $1 \cdot \ell_{ret}$ cell has predecessors $\ell_1 \cdot \ell_{ret}$ and ℓ_1, we issue requests for the values of those cells. Cell ℓ_1 is empty, but depends on $\ell_0 \cdot \ell_1$ and ℓ_0, so more requests are issued. Both of those cells hold values, so we can compute and store the value of ℓ_1. Now, having satisfied its dependencies, we can compute the value of $1 \cdot \ell_{ret}$, as shown in Fig. 4a. Note that DAIGs are always acyclic, so this recursive traversal of dependencies is well-founded.

Crucially, these results are now *memoized* for future incremental reuse; a subsequent query for ℓ_{ret}, for example, will memo match on $1 \cdot \ell_{ret}$ and only need to compute $2 \cdot \ell_{ret}$ and its dependencies from scratch. This fine-grained reuse of intermediate abstract interpretation results is a key feature of the DAIG encoding for demand-driven analysis.

To handle developer edits to code, DAIGs are also naturally *incremental*, efficiently recomputing and reusing analysis results across multiple program versions, following the incremental computation with names approach [20]. Consider a program edit which adds a logging statement `print("p is null")` just before the `return` at ℓ_1 in Fig. 1. Intuitively, program behaviors are unchanged at those locations unreachable from the added statement, so an incremental analysis should only need to re-analyze the sub-DAIG reachable from the new statement.

Fig. 4b illustrates this program edit's effect on the DAIG. The green nodes corresponding to the added statement cell $\ell_1 \cdot \ell_7$ and its corresponding abstract state cell ℓ_7.Nodes forward-reachable from the green nodes — those marked in red — are invalidated (a.k.a. "dirtied") by our incremental computation engine. In particular, cells $\ell_1 \cdot \ell_{ret}$ and ℓ_{ret} containing abstract states φ' and φ'', respectively, are dirtied.

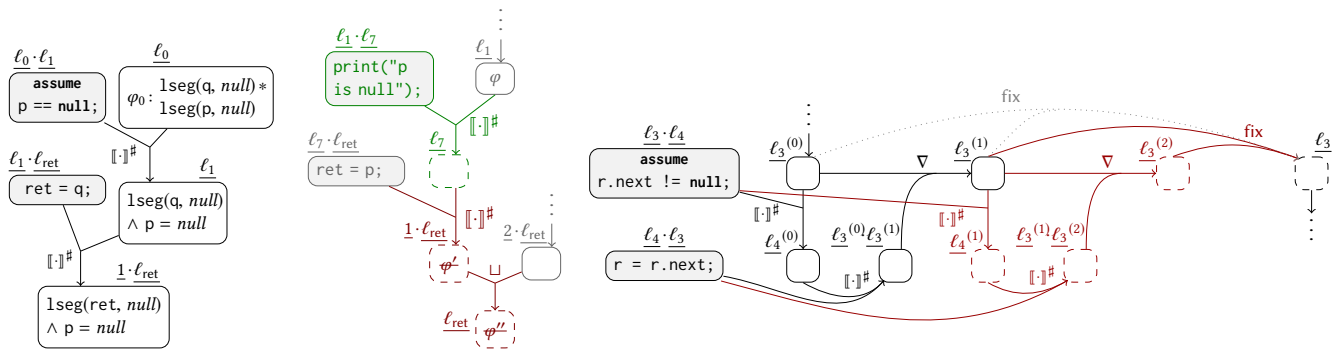
Crucially, while nodes are dirtied *eagerly*, they are recomputed from up-to-date inputs *lazily*, only when demanded. That is, the DAIG encoding allows our analysis to avoid constant recomputation of an analysis as a program is edited, instead computing results on demand while soundly keeping track of which intermediate results — possibly from a previous program version — are available for reuse. For example, assuming that ℓ_1 and $2 \cdot \ell_{ret}$ were both computed before the edit, a query for ℓ_{ret} must execute only two transfers and one join: the red and green edges of Fig. 4b. This represents a significant savings over recomputing the entire analysis — including the loop fixed point — as would be necessary without incremental analysis.

If desired, an auxiliary memoization (memo) table can also be used to cache computations independent of program locations to enable further incrementalization, with names based on the input values (e.g., memoizing $[\![s_0]\!]^\sharp(\varphi_0)$ in a cell named $[\![\cdot]\!]^\sharp \cdot s_0 \cdot \varphi_0$). As with batch analysis, it is sound to drop cached results from the DAIG and/or memo table and later recompute those results if needed, trading efficiency of reuse for a lower memory footprint.

2.3 Cyclic Control Flow and Demanded Fixed Points

As shown in Section 2.1, encoding program structure and analysis data-flow into a dependency graph is relatively straightforward when the control-flow graph is acyclic. However, when handling loops or recursion, like lines ℓ_3 and ℓ_4 in Fig. 1, an abstract interpreter's fixed-point computation is inherently cyclic. Properly handling these cyclic control-flow and data-flow dependency structures is the crux of realizing demanded abstract interpretation. For instance, introducing cyclic dependencies in the DAIG yields an unclear evaluation

[3] That is, a sequence of iterated `next` pointer dereferences from x to y.



(a) Demand-driven query **(b)** Incremental edit **(c)** Cyclic-control flow and demanded fixed points

Figure 4. Demanded abstract interpretation: (a) Demanding a value for $\underline{1 \cdot \ell_{\text{ret}}}$ recursively triggers demand for its dependencies and is resolved by computing its value from the statements in $\underline{\ell_0 \cdot \ell_1}$ and $\underline{\ell_1 \cdot \ell_{\text{ret}}}$ and the initial state φ_0 in $\underline{\ell_0}$. We show only the relevant subgraph here, but this operation occurs in the full DAIG of Fig. 3. (b) DAIG from Fig. 3 updated to reflect nodes added ($\underline{\ell_1 \cdot \ell_7}$ and $\underline{\ell_7}$) and potentially affected ($\underline{1 \cdot \ell_{\text{ret}}}$ and $\underline{\ell_{\text{ret}}}$) by the edit in Section 2.2. All other nodes are unchanged. (c) DAIG for the ℓ_3-to-ℓ_4-to-ℓ_3 loop of Fig. 2 after one demanded unrolling, with the new DAIG region shown in red and the (removed) pre-unrolling fix edge shown in dotted grey. Note that cells containing program syntax are not duplicated. The DAIG with the black vertices and edges along with the grey edge (but not the red ones) is the initial sub-DAIG for the dashed ellipse in Fig. 3.

semantics. The key insight we leverage is that we can instead enrich the demand-driven query evaluation and the incremental edit semantics to dynamically evolve DAIGs such that each step preserves the acyclic dependency structure invariant. To do so, we use a distinguished edge label (fix) to indicate a dependency on the fixed-point of a given region of the DAIG, which is then dynamically unrolled on-demand by query evaluation and rolled by incremental edits.

The details are formalized in Section 5; we proceed here by example on the demanded abstract interpretation of the append program of Fig. 2. Consider Fig. 4c, and focus on the black and grey vertices and edges, ignoring the red for the moment. Rather than encoding the CFG back edge from ℓ_4 to ℓ_3 directly into the DAIG (violating acyclicity in the process), the ℓ_3-to-ℓ_4-to-ℓ_3 loop is initially encoded with separate reference cells for the 0th and 1st abstract iterates at the loop head ℓ_3 (named $\underline{\ell_3}^{(0)}$ and $\underline{\ell_3}^{(1)}$ respectively), along with a fix edge from those two cells to ℓ_3's fixed-point cell $\underline{\ell_3}$, as seen in the grey dotted edge. Crucially, this initial DAIG is acyclic.

Query Evaluation. For query evaluation, we can compute fixed points on demand by "unrolling" the *abstract interpretation* of loop bodies in the DAIG one *abstract* iteration at a time until a fixed point is reached, preserving the acyclic DAIG invariant at each step. That is, our key observation is to unroll at the semantic level of the abstract interpretation rather than the syntactic level of the control-flow graph.

From the initial DAIG in Fig. 4c, when the fixed-point $\underline{\ell_3}$ is demanded, its dependencies — the 0th and 1st abstract iterates — are computed. If their values are equal, then a fixed-point has been reached and may be written to $\underline{\ell_3}$. If their values are *not* equal, then the abstract interpretation of the loop body — but not the loop body statements — is unrolled one step further and the fix edge slides forward to now depend on the 1st and 2nd abstract iterates, as seen in the red cells and edges of Fig. 4c. And crucially, this one-step unrolled DAIG is also acyclic.

From here, the process continues, and termination is guaranteed by leveraging the standard argument of abstract interpretation meta-theory: the sequence of abstract iterates in the cells $\underline{\ell_3}^{(0)}, \underline{\ell_3}^{(1)}, \underline{\ell_3}^{(2)}, \ldots$ converges because it is produced by widening a monotonically increasing sequence of abstract states, so this demanded unrolling of fix occurs only finitely — but unboundedly — many times. We see that in essence, the sequence of abstract interpretation iterates $\underline{\ell_3}^{(0)}, \underline{\ell_3}^{(1)}, \underline{\ell_3}^{(2)}, \ldots$ are encoded into the DAIG on demand during query evaluation.

In classical abstract interpretation, a widen ∇ is a join that enforces convergence during interpretation and thus is only strictly needed if the abstract domain has infinite height. Our approach can be seen as an application of this widening principle to demanded computation. For an abstract domain of finite height k, it would have been sufficient to encode the unrolling of fix eagerly into an acyclic DAIG by inlining the abstract iteration k times to k iterate cells $\underline{\ell_3}^{(0)}, \ldots, \underline{\ell_3}^{(k)}$. However, many expressive, real-world abstract domains — including the shape analysis domain of our example and most numerical domains — are of infinite height.

Incremental Edits. Since the acyclic DAIG invariant is always preserved, invalidating on incremental edits still only requires eager dirtying forwards in the DAIG, with some special semantics for fix edges. When dirtying along a fix

statements	$s \in \mathit{Stmt}$	
locations	$\ell \in \mathit{Loc}$	
control-flow edges	$e \in \mathit{Edge}$	$::= \ell \dashv[s]\vdash \ell'$
programs	$\langle L, E, \ell_0 \rangle$	$: \mathcal{P}(\mathit{Loc}) \times \mathcal{P}(\mathit{Edge}) \times \mathit{Loc}$
concrete states	$\sigma \in \Sigma$	(with initial state σ_0)
concrete semantics	$[\![\cdot]\!]$	$: \mathit{Stmt} \to \Sigma \to \Sigma_\perp$
collecting semantics	$[\![\cdot]\!]^*_{\langle L, E, \ell_0 \rangle}$	$: \mathit{Loc} \to \mathcal{P}(\Sigma)$

Figure 5. A generic programming language of control-flow graphs edge-labelled by an unspecified statement language.

edge, the fix edge is rolled back to a non-dirty cell (i.e., the 0th and 1st iterate). In Fig. 4c, if the statement cell $\ell_4 \cdot \ell_3$ is edited, then dirtying will happen along the red solid fix edge at which point it will slide back to be the grey dotted one.

Interprocedural Demand. This demanded unrolling of fix also suggests an approach to interprocedural demanded abstract interpretation parameterized by a context-sensitivity policy. To analyze a call when evaluating a query, we construct a DAIG for the callee procedure on demand, indexed by a context determined opaquely by the context-sensitivity policy. Then, invalidation on incremental edits is as expected: eagerly dirtying forwards and potentially into callee DAIGs.

3 Preliminary Definitions

Our technique lifts a program and an abstract interpreter *together* into a demanded abstract interpretation graph (DAIG), a representation that is amenable for sound incremental and demand-driven program analysis. By design, this construction is *generic* in the underlying programming language and concrete semantics as well as the abstract domain and abstract semantics. In this section, we fix a generic programming language and an abstract interpreter interface that serve as inputs to DAIG construction, both to define syntax and to make explicit our assumptions about their semantic properties. Selected instantiations of the framework for real-world analysis problems are given in Section 7.

Programs under analysis are given as control-flow graphs, edge-labelled by an unspecified statement language and interpreted by a denotational concrete semantics as shown in Fig. 5. A program $\langle L, E, \ell_0 \rangle$ is a 3-tuple composed of a set L of control locations, a set E of directed, statement-labelled control-flow edges between locations, and an initial location ℓ_0. We say that a program $\langle L, E, \ell_0 \rangle$ is *well-formed* when (1) ℓ_0 and all locations in E are drawn from L, and (2) L and E form a reducible control-flow graph. These conditions ensure that we avoid degenerate edge cases and only consider control flow graphs which correspond to realistic programs [4].

Statements are interpreted by the concrete denotational semantics $[\![\cdot]\!]$ as partial functions over concrete program states. As is standard, we can also lift this statement semantics to a collecting $[\![\cdot]\!]^*_{\langle L, E, \ell_0 \rangle}$ of full programs, by computing the transitive closure of the statement semantics over a flow graph. That is, $[\![\ell]\!]^*_{\langle L, E, \ell_0 \rangle}$ is the set of all concrete states that can be witnessed at program location ℓ in a valid program execution. We elide the subscript when it is clear from context. Such a collecting semantics is uncomputable in general, but is an important tool for reasoning about analysis soundness.

Now, we define the interface of a generic abstract interpreter over this control-flow graph language. These definitions are intended simply to fix notation and minimize ambiguity and are as standard as possible.

An abstract interpreter is a 6-tuple $\langle \Sigma^\sharp, \varphi_0, [\![\cdot]\!]^\sharp, \sqsubseteq, \sqcup, \nabla \rangle$ composed of:

- An *abstract domain* $\Sigma^\sharp$ (elements of which are referred to as *abstract states*) which forms a semi-lattice under
  - a *partial order* $\sqsubseteq \in \mathcal{P}(\Sigma^\sharp \times \Sigma^\sharp)$ with a bottom $\perp \in \Sigma^\sharp$
  - a *least upper bound* (a.k.a. *join*) $\sqcup : \Sigma^\sharp \to \Sigma^\sharp \to \Sigma^\sharp$
- An *initial abstract state* $\varphi_0 \in \Sigma^\sharp$
- An *abstract semantics* $[\![\cdot]\!]^\sharp : \mathit{Stmt} \to \Sigma^\sharp \to \Sigma^\sharp$ that interprets program statements as monotone functions over abstract states.
- A *widening* operator $\nabla : \Sigma^\sharp \to \Sigma^\sharp \to \Sigma^\sharp$ that is an upper bound operator (i.e., $(\varphi \sqcup \varphi') \sqsubseteq (\varphi \nabla \varphi')$ for all φ, φ') and enforces convergence (i.e., for all increasing sequences of abstract states $\varphi_0 \sqsubseteq \varphi_1 \sqsubseteq \varphi_2 \sqsubseteq \cdots$, the sequence $\varphi_0,\ \varphi_0 \nabla \varphi_1,\ (\varphi_0 \nabla \varphi_1) \nabla \varphi_2,\ \ldots$ converges).

Furthermore, a concretization function $\gamma : \Sigma^\sharp \to \mathcal{P}(\Sigma)$ gives meaning to abstract states. We say that a concrete state σ *models* an abstract state φ (equivalently, that φ *abstracts* σ), written $\sigma \models \varphi$, when $\sigma \in \gamma(\varphi)$).

Definition 3.1 (Local Abstract Interpreter Soundness)**.** An abstract interpreter $\langle \Sigma^\sharp, \varphi_0, [\![\cdot]\!]^\sharp, \sqsubseteq, \sqcup, \nabla \rangle$ is locally sound if for all σ, φ, s, if $\sigma \models \varphi$ and $[\![s]\!]\sigma \neq \perp$ then $[\![s]\!]\sigma \models [\![s]\!]^\sharp \varphi$.

Local soundness can be extended to a global soundness property: if the abstract semantics are locally sound, then the abstract interpreter computes a sound over-approximation of the possible concrete states at each location.

Proposition 3.2 (Global Abstract Interpreter Soundness)**.** *If* $\langle \Sigma^\sharp, \varphi_0, [\![\cdot]\!]^\sharp, \sqsubseteq, \sqcup, \nabla \rangle$ *is locally sound and* $\sigma_0 \models \varphi_0$ *then it induces an abstract collecting semantics* $[\![\cdot]\!]^{\sharp *}_{\langle L, E, \ell_0 \rangle} : \mathit{Loc} \to \Sigma^\sharp$ *such that for all* $\sigma \in [\![\ell]\!]^*_{\langle L, E, \ell_0 \rangle}$, $\sigma \models [\![\ell]\!]^{\sharp *}_{\langle L, E, \ell_0 \rangle}$.

We elide the abstract collecting semantics $[\![\cdot]\!]^{\sharp *}_{\langle L, E, \ell_0 \rangle}$; it is similarly a transitive closure of the abstract statement semantics over a flow graph. It is a well-known result that global abstract interpreter soundness is implied by local soundness [11] and that such a global fixed-point is computable using the chaotic iteration method with widening [8].

4 Demanded AI Graphs

Recall from Section 2 that a demanded abstract interpretation graph (DAIG) is a directed acyclic hypergraph, whose vertices are reference cells containing program syntax or intermediate analysis results, and whose edges reflect analysis

functions	f	$::= [\![\cdot]\!]^\sharp \mid \sqcup \mid \nabla \mid \mathsf{fix}$
values	v	$::= s \mid \varphi$
names	$n \in Nm$	$::= \underline{\ell} \mid \underline{f} \mid \underline{i} \mid \underline{v} \mid n_1 \cdot n_2 \mid n^{(i)}$
types	τ	$\in \{Stmt,\ \Sigma^\sharp\}$
reference cells	$r \in Ref$	$::= n[v : \tau] \mid n[\varepsilon : \tau]$
computations	$c \in Comp$	$::= n \leftarrow f(n_1, \ldots, n_k)$
DAIGs	$\mathcal{D}$	$: \mathcal{P}(Ref) \times \mathcal{P}(Comp)$

Figure 6. Demanded Abstract Interpretation Graphs, edge-labelled by analysis functions and connecting named reference cells storing statements and abstract states.

dataflow relationships among those cells. In Fig. 6, we show a syntax for DAIGs. A DAIG $\mathcal{D} = \langle R, C\rangle$ is composed of a set $R \subseteq Ref$ of named reference cells connected by computation edges $C \subseteq Comp$. A computation $c\colon n \leftarrow f(n_1, \ldots, n_k)$ is an edge connecting sources $\{n_1, \ldots, n_k\}$ to a singleton destination $\{n\}$, labeled by some analysis function f.

Names $\underline{\ell}$, $\underline{f}$, $\underline{v}$ and $\underline{i}$ correspond respectively to locations ℓ, functions f, values v and integers i, supporting memoization of those syntactic constructs. Name products $n_1 \cdot n_2$ support the construction of more complicated names, and i-primed names $n^{(i)}$ allow variants of a single name to be distinguished as loops are unrolled: $n^{(i)}$ is the ith unrolled copy of the name n in a loop. All name equalities are decided structurally.

Values include statements s and abstract states $\varphi \in \Sigma^\sharp$. Reference cells bind names to values or the absence thereof (denoted ε), while computations specify analysis data-flow dependencies between reference cells.

We denote by $\mathcal{D}[n \mapsto v]$ the DAIG identical to $\mathcal{D}$ except that the reference cell named n now holds value v. We also denote DAIG reachability by $n \rightsquigarrow_{\mathcal{D}} n'$ (eliding the subscript when it is clear from context), and define helper functions `name`, `srcs`, and `dest` to project out, respectively, the name n of a reference cell $n[v_\varepsilon : \tau]$ and the source names $\{n_1, \ldots, n_k\}$ or destination name n of a computation $n \leftarrow f(n_1, \ldots, n_k)$. Finally, the typing judgment $R \vdash n \leftarrow f(n_1, \ldots, n_k)$ holds when n_1 through n_k name references in R with the same types as f's inputs and n names a reference in R with the same type as f's output.

Definition 4.1 (DAIG Well-formedness). A DAIG $\mathcal{D} = \langle R, C\rangle$ is subject to the following well-formedness constraints.

(1) References are named uniquely:
$$\forall r, r' \in R\ .\ \mathsf{name}(r) = \mathsf{name}(r') \iff r = r'$$
(2) Computations have unique destinations:
$$\forall c, c' \in C\ .\ \mathsf{dest}(c) = \mathsf{dest}(c') \iff c = c'$$
(3) Dependencies are acyclic: $\nexists\ r \in R\ .\ \mathsf{name}(r) \rightsquigarrow \mathsf{name}(r)$
(4) Computations are well-typed with respect to references:
$$\forall c \in C\ .\ R \vdash c$$
(5) Empty references have dependencies:
$$\forall n[\varepsilon : \tau] \in R\ .\ \exists\ c \in C\ .\ n = \mathsf{dest}(c)$$

Beyond these basic well-formedness conditions, a DAIG's structure must also properly encode an abstract interpretation computation over an underlying program. Given a program's CFG and an abstract interpreter interface (as defined in Section 3), there are three general cases shown in Fig. 7 to consider when examining a corresponding DAIG. The key property is that demand-driven query evaluation and incremental edits will evolve the DAIG but preserve the following consistency conditions:

(1) A forward CFG edge $\ell' \dashv[s]\mapsto \ell$ to a *non-join* location is encoded by a transfer function-labelled DAIG edge, connecting reference cells for its abstract pre-state (named $n_{\ell'}$) and statement label (named $\underline{\ell'} \cdot \underline{\ell}$) to a reference cell for its abstract post-state (named n_ℓ).[4]
(2) Forward CFG edges to a *join* location ℓ are a bit more complex, introducing intermediate cells to encode the join $\sqcup$ into the DAIG. For each incoming edge to ℓ with statement s_i, we introduce a three-cell transfer function construct similar to case (1), with an output cell $\underline{i} \cdot n_\ell$ named uniquely for that edge. Then, a single join edge connects each pre-join abstract state $\underline{i} \cdot n_\ell$ to n_ℓ, the abstract post-state at ℓ.
(3) As described informally in Section 2.3, our framework analyzes CFG back edges by unrolling the abstract fixed-point computation to evolve DAIGs on demand, so this diagram is parameterized by a number k of such unrollings. Given the CFG back edge $\ell' \dashv[s]\mapsto \ell$, a transfer function DAIG edge connects the abstract state after one abstract iteration (named $\underline{\ell'}^{(0)}$) and s to a pre-widen abstract state at the loop head ℓ (named $\underline{\ell}^{(0)} \cdot \underline{\ell}^{(1)}$), which is connected with the previous abstract state at the loop head ($\underline{\ell}^{(0)}$) to the next ($\underline{\ell}^{(1)}$) via a widen edge.[5] This acyclic structure is repeated k times in the DAIG (with $k = 1$ in the initial construction and further unrollings generated on demand as decribed in Section 5), thereby encoding the unbounded fixed-point computation. Lastly, the fix edge — indicating a dependency on the eventual fixed point — connects the two greatest abstract iterates to the reference cell ($\underline{\ell}$) for the fixed-point abstract state at the loop head.

Definition 4.2 (DAIG–CFG Consistency). A DAIG $\mathcal{D} = \langle R, C\rangle$ is consistent with a program CFG $\langle L, E, \ell_0\rangle$, written $\mathcal{D} \cong \langle L, E, \ell_0\rangle$, when it is well-formed and its structure accurately encodes the abstract interpretation of that program.

A formal statement of the $\mathcal{D} \cong \langle L, E, \ell_0\rangle$ relation is given in the appendix, closely following the structure of the above description and Fig. 7.

The above establishes when the structure of a DAIG is consistent with the program's CFG. A DAIG is consistent

[4] We write n_ℓ for the name of the abstract state at ℓ throughout this section: $\underline{\ell}^{(0)}$ if ℓ belongs to any natural loop and $\underline{\ell}$ otherwise. Loop heads ℓ are a special case: n_ℓ is $\underline{\ell}^{(0)}$ (the abstract state at loop entry) when the destination of a DAIG edge and $\underline{\ell}$ (the fixed point at ℓ) otherwise.
[5] We fix the widening strategy of applying ∇ every iteration for simplicity, but the same general idea applies when it is applied intermittently

with an abstract interpret*ation* of the program when the partial analysis results stored in the DAIG are consistent with the partial abstract interpretation, or formally as follows:

Definition 4.3 (DAIG–AI Consistency). A DAIG $\mathcal{D} = \langle R, C\rangle$ is consistent with an abstract interpreter $\langle \Sigma^\sharp, \varphi_0, [\![\cdot]\!]^\sharp, \sqsubseteq, \sqcup, \nabla\rangle$, written $\mathcal{D} \cong \langle \Sigma^\sharp, \varphi_0, [\![\cdot]\!]^\sharp, \sqsubseteq, \sqcup, \nabla\rangle$, when all partial analysis results stored in R are consistent with the computations encoded by C, such that $\underline{\ell_0}[\varphi_0 : \Sigma^\sharp] \in R$ and

$$\forall\, n[v : \Sigma^\sharp] \in R\,,\ n \leftarrow f(n_1, \ldots, n_k) \in C\,.$$
$$\{n_i[v_i : \tau_i] \mid 1 \le i \le k\} \subseteq R \wedge \begin{cases} v = v_1 = v_2 & \text{if } f = \text{fix} \\ v = f(v_1, \ldots, v_k) & \text{otherwise} \end{cases}$$

Given a program's CFG and a generic abstract interpretation interface, we can construct an initial DAIG that is consistent with a classical (batch) abstract interpretation:

Lemma 4.1 (Initial DAIG Construction, Well-Formedness, CFG-Consistency, and AI-Consistency). *There exists a constructive procedure $\mathcal{D}_{\text{init}}$ such that for all well-formed programs $\langle L, E, \ell_0\rangle$, the initial DAIG*

$$\mathcal{D} = \mathcal{D}_{\text{init}}(\langle L, E, \ell_0\rangle, \langle \Sigma^\sharp, \varphi_0, [\![\cdot]\!]^\sharp, \sqsubseteq, \sqcup, \nabla\rangle)$$

is well formed and consistent with both the target program (i.e. $\mathcal{D} \cong \langle L, E, \ell_0\rangle$) and underlying abstract interpreter (i.e. $\mathcal{D} \cong \langle \Sigma^\sharp, \varphi_0, [\![\cdot]\!]^\sharp, \sqsubseteq, \sqcup, \nabla\rangle$).

Proof sketch. We define such a $\mathcal{D}_{\text{init}}$ in the appendix and show that it produces well-formed results consistent with both the target program and the underlying abstract interpreter. Its definition tracks closely with the informal and diagrammatic descriptions above, constructing DAIG structures that correspond to the input CFG. □

5 Demanded AI by Evaluating DAIGs

In this section, we give an operational semantics for demand-driven and incremental evaluation of DAIGs. A state in the operational semantics consists of a DAIG $\mathcal{D}$ and also an auxiliary memoization table M, which can be used to reuse previously-computed analysis results independent of program location. Memoization tables are finite maps from names n to abstract states $\varphi \in \Sigma^\sharp$, and we write $M(n)$ for the abstract state mapped to by n in M, $\text{dom}(M)$ for the set of names in the domain of M, and $M[n \mapsto \varphi]$ for the extension of M with a new mapping from n to φ.

The DAIG operational semantics are split into two judgments, corresponding to *queries* and *edits* over both analysis results and program syntax. Both interaction modes are given in a small-step style, describing the effects of each operation on the DAIG and auxiliary memo table.

5.1 Query Evaluation Semantics

A query for the value of the reference cell with name n, given some initial DAIG $\mathcal{D}$ and auxiliary memo table M, yields a value v and (possibly unchanged) DAIG and memo-table structures $\mathcal{D}'$ and M'. This operation is defined inductively

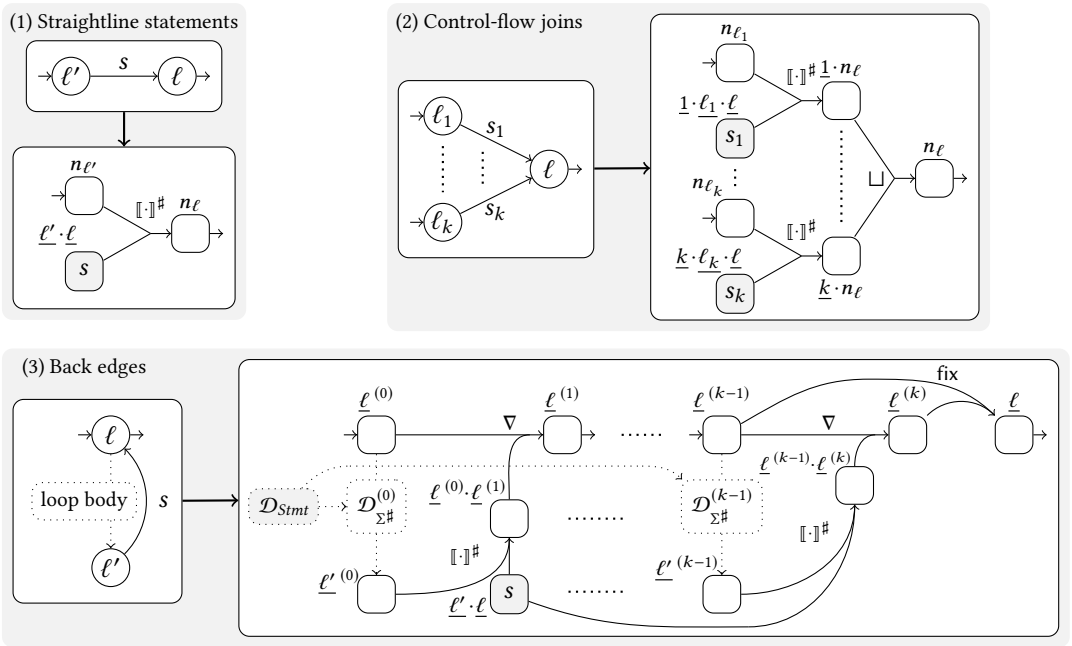


Figure 7. DAIG–CFG Consistency (Definition 4.2) in diagram form, illustrating how different CFG structures are encoded into DAIG structures. In subfigure (3), we apply some ad-hoc shorthands for the DAIG encoding of the loop body: $\mathcal{D}_{Stmt}$ contains all of its statement reference cells, while $\mathcal{D}^{(i)}_{\Sigma^\sharp}$ contains all of its abstract state reference cells, with iteration counts set to i. Each dotted line from D_{Stmt} thus represents one or more DAIG edges, from each statement to corresponding abstract states.

$$\boxed{\mathcal{D}, M \vdash n \Rightarrow v \,;\, \mathcal{D}', M'}$$

Q-Reuse

$$\frac{n[v : \tau] \in R}{\langle R, C\rangle, M \vdash n \Rightarrow v \,;\, \langle R, C\rangle, M}$$

Q-Match

$$\frac{\begin{array}{c}\mathcal{D}_0 = \langle R, C\rangle \qquad n[\varepsilon : \tau] \in R \qquad n \leftarrow f(n_1, \ldots, n_k) \in C \\ \mathcal{D}_{i-1}, M_{i-1} \vdash n_i \Rightarrow v_i \,;\, \mathcal{D}_i, M_i \quad (\text{for } i \in [1, k]) \\ \underline{f}\cdot(\underline{v_1}\cdots\underline{v_k}) \in \mathsf{dom}(M_k) \qquad v = M_k(\underline{f}\cdot(\underline{v_1}\cdots\underline{v_k}))\end{array}}{\mathcal{D}_0, M_0 \vdash n \Rightarrow M_k(\underline{f}\cdot(\underline{v_1}\cdots\underline{v_k})) \,;\, \mathcal{D}_k[n \mapsto M_k(\underline{f}\cdot(\underline{v_1}\cdots\underline{v_k}))], M_k}$$

Q-Miss

$$\frac{\begin{array}{c}\mathcal{D}_0 = \langle R, C\rangle \qquad n[\varepsilon : \tau] \in R \qquad n \leftarrow f(n_1, \ldots, n_k) \in C \\ \mathcal{D}_{i-1}, M_{i-1} \vdash n_i \Rightarrow v_i \,;\, \mathcal{D}_i, M_i \quad (\text{for } i \in [1, k]) \\ \underline{f}\cdot(\underline{v_1}\cdots\underline{v_k}) \notin \mathsf{dom}(M_k) \qquad v = f(v_1, \ldots, v_k) \qquad f \neq \mathsf{fix}\end{array}}{\mathcal{D}_0, M_0 \vdash n \Rightarrow v \,;\, \mathcal{D}_k[n \mapsto v], M_k[\underline{f}\cdot(\underline{v_1}\cdots\underline{v_k}) \mapsto v]}$$

Q-Loop-Converge

$$\frac{\begin{array}{c}n[\varepsilon : \tau] \in R \qquad n \leftarrow \mathsf{fix}(n_1, n_2) \in C \\ \langle R, C\rangle, M \vdash n_1 \Rightarrow v \,;\, \mathcal{D}', M' \qquad \mathcal{D}', M' \vdash n_2 \Rightarrow v \,;\, \mathcal{D}'', M''\end{array}}{\langle R, C\rangle, M \vdash n \Rightarrow v \,;\, D''[n \mapsto v], M''}$$

Q-Loop-Unroll

$$\frac{\begin{array}{c}n[\varepsilon : \tau] \in R \qquad c = n \leftarrow \mathsf{fix}(\underline{\ell}^{(k-1)}, \underline{\ell}^{(k)}) \in C \\ \langle R, C\rangle, M \vdash \underline{\ell}^{(k-1)} \Rightarrow v' \,;\, \mathcal{D}', M' \\ \mathcal{D}', M' \vdash \underline{\ell}^{(k)} \Rightarrow v'' \,;\, \mathcal{D}'', M'' \\ v' \neq v'' \qquad \mathsf{unroll}(\mathcal{D}'', c), M'' \vdash n \Rightarrow v \,;\, \mathcal{D}''', M'''\end{array}}{\langle R, C\rangle, M \vdash n \Rightarrow v \,;\, \mathcal{D}''', M'''}$$

Figure 8. Operational semantics rules governing *queries* for the contents of a DAIG. The judgment form $\mathcal{D}, M \vdash n \Rightarrow v \,;\, \mathcal{D}', M'$ is read as "Requesting n from DAIG $\mathcal{D}$ with auxiliary memo table M yields value v, updated DAIG $\mathcal{D}'$, and updated memo table M'."

by the $\mathcal{D}, M \vdash n \Rightarrow v \,;\, \mathcal{D}', M'$ judgment form, whose inference rules are given in Fig. 8.

There are two potential ways to reuse previously-computed analysis results: either in DAIG $\mathcal{D}$ or the auxiliary memo table M. The Q-Reuse rule handles the case where the DAIG cell named by n already holds a value, returning that value and leaving the DAIG and memo table unchanged.

The Q-Match and Q-Miss rules handle the case where n is empty in $\mathcal{D}$. In both cases, queries are issued for the input cells n_1 through n_k to the computation f that outputs to n. In Q-Match, the auxiliary memo table is matched: f has already been computed for the relevant inputs, so the result is retrieved from M_k, the memo table after querying the input cells, and stored in n (i.e., via $\mathcal{D}_k[n \mapsto M_k(\underline{f}\cdot(\underline{v_1}\cdots\underline{v_k}))]$). Note that this notation is a low-level write to the reference cell named by n, not an external edit that would trigger invalidation (which we will describe below in Section 5.3).

Q-Miss handles memo table misses by computing and memoizing $f(v_1, \ldots, v_k)$ before storing the result in both the DAIG $\mathcal{D}$ and the auxiliary memo table M (i.e., at names n and $\underline{f}\cdot(\underline{v_1}\cdots\underline{v_k})$, respectively).

5.2 Demanded Fixed Points

Demanded unrolling is the process by which we compute abstract interpretation fixed-points over cyclic control flow graphs without introducing cyclic dependencies into DAIGs, as described informally in Section 2 and represented graphically in Fig. 4c. The semantics are formalized by the Q-Loop-Converge and Q-Loop-Unroll rules in Fig. 8.

Recall that fix is a special function symbol indicating an analysis fixed-point computation. The destination of a fix edge is a loop-head cell for storing a fixed-point invariant, and its sources are the two greatest abstract iterates of said loop head yet computed. When those abstract iterates have the same value v, the analysis has reached a fixed point, and a query for the fixed point may return v. However, when they are unequal, a query triggers an unrolling in the DAIG: the loop body's abstract state reference cells are unrolled one more iteration, the fix edge's sources are shifted forward one iteration, and then the fixed-point query is reissued.

This procedure differs from concrete loop unrolling — as applied, e.g., by an optimizing compiler or bounded model checker — in that it applies to the DAIG's reified abstract interpretation computation, including joins and widens, *not* to the concrete syntax of the program under analysis. As a result, it is sound with respect to the concrete semantics of the program under analysis and is guaranteed to converge.

As the name suggests, Q-Loop-Converge applies when the abstract interpretation has reached a fixed point. Since the dependencies of the fix edge, n_1 and n_2, are consecutive abstract iterates at the head of the corresponding loop, their evaluation to the same value v indicates that loop analysis has converged, so v may be stored in the DAIG and returned.

On the other hand, Q-Loop-Unroll applies when the abstract interpretation has not yet reached a fixed point, since the two most recent abstract iterates are unequal. In this case, the loop is unrolled once by the unroll helper function and the query for the fixed point is reissued. The unroll helper function used in Q-Loop-Unroll takes a DAIG $\mathcal{D}$ and a fix edge and unrolls the loop corresponding to the fix edge by one iteration in $\mathcal{D}$. It is defined as follows:

$$\begin{array}{l}\mathsf{unroll}\left(\langle R, C\rangle, c = \underline{\ell} \leftarrow \mathsf{fix}(\underline{\ell}^{(k-1)}, \underline{\ell}^{(k)})\right) \triangleq \langle R', C'\rangle, \text{ where} \\ \quad R' = R \cup \left\{\mathsf{incr}(n)[\varepsilon : \Sigma^\sharp] \;\middle|\; \underline{\ell}^{(k-1)} \rightsquigarrow n \rightsquigarrow \underline{\ell}^{(k)}\right\} \text{ and} \\ \quad C' = C/\{c\} \cup \left\{\underline{\ell} \leftarrow \mathsf{fix}\left(\underline{\ell}^{(k)}, \underline{\ell}^{(k+1)}\right)\right\} \\ \qquad\qquad \cup \left\{\mathsf{incr\text{-}c}(c) \;\middle|\; \underline{\ell}^{(k-1)} \rightsquigarrow \mathsf{dest}(c) \rightsquigarrow \underline{\ell}^{(k)}\right\}\end{array}$$

where incr and incr-c increment the iteration counts of names. Intuitively, unroll takes the region of the DAIG forwards-reachable from the $k{-}1^{th}$ abstract iterate $\underline{\ell}^{(k-1)}$ and backwards-reachable from the k^{th} abstract iterate $\underline{\ell}^{(k)}$ and duplicates it

$$\boxed{\mathcal{D} \vdash n \Leftarrow v_\varepsilon \,;\, \mathcal{D}'}$$

E-Commit

$$\frac{\begin{array}{c}\forall c \in C \,.\, n \in \mathsf{srcs}(c) \implies \mathsf{dest}(c)[\varepsilon : \tau] \in R \\ \underline{\ell} \leftarrow \mathsf{fix}(\underline{\ell}^{(i)}, \underline{\ell}^{(i+1)}) \in C \implies \underline{\ell}^{(1)}[\varepsilon : \tau] \in R \\ v_\varepsilon = \varepsilon \implies \exists c \in C \,.\, n = \mathsf{dest}(c)\end{array}}{\langle R, C\rangle \vdash n \Leftarrow v_\varepsilon \,;\, \langle R, C\rangle[n \mapsto v_\varepsilon]}$$

E-Propagate

$$\frac{n \rightsquigarrow_{\mathcal{D}} n' \qquad \mathcal{D} \vdash n' \Leftarrow \varepsilon \,;\, \mathcal{D}' \qquad \mathcal{D}' \vdash n \Leftarrow v_\varepsilon \,;\, \mathcal{D}''}{\mathcal{D} \vdash n \Leftarrow v_\varepsilon \,;\, \mathcal{D}''}$$

E-Loop

$$\frac{\begin{array}{c}\underline{\ell} \leftarrow \mathsf{fix}(\underline{\ell}^{(k-1)}, \underline{\ell}^{(k)}) \in C \qquad \langle R, C'\rangle \vdash \underline{\ell}^{(1)} \Leftarrow \varepsilon \,;\, \mathcal{D}' \\ C' = C / \left\{\underline{\ell} \leftarrow \mathsf{fix}(\underline{\ell}^{(k-1)}, \underline{\ell}^{(k)})\right\} \cup \left\{\underline{\ell} \leftarrow \mathsf{fix}(\underline{\ell}^{(0)}, \underline{\ell}^{(1)})\right\}\end{array}}{\langle R, C\rangle \vdash \underline{\ell}^{(k)} \Leftarrow \varepsilon \,;\, \mathcal{D}'}$$

Figure 9. Operational semantics rules governing *edits* to the contents of a DAIG. The judgment $\mathcal{D} \vdash n \Leftarrow v_\varepsilon \,;\, \mathcal{D}'$ is read as "Mutating reference cell n with value v_ε of a DAIG $\mathcal{D}$ yields updated DAIG $\mathcal{D}'$," where v_ε ranges over values v and the "empty" symbol ε.

while incrementing all name's iterations counts from $k-1$ to k, then shifts the fix edge forward one iteration. Crucially, this operation preserves the DAIG acyclicity invariant.

5.3 Incremental Edit Semantics

An edit to a DAIG $\mathcal{D}$ occurs when a value v is written to some reference cell named n in $\mathcal{D}$ by an external mutator. This edit must both update n and also clear the value of (or "dirty") any reference cell that (transitively) depends on n. Here we give a full definition of the edit operation, using the $\mathcal{D} \vdash n \Leftarrow v_\varepsilon \,;\, \mathcal{D}'$ judgment given in Fig. 9.

As described informally in Section 2, invalidation proceeds by dirtying forwards in the acyclic DAIG, except that the implicit cyclic dependency from fix edges must be accounted for, by rolling back to a non-dirty source cell.

The E-Commit rule is a base case: if the edited cell's downstream dependencies are all empty, then the edit may be performed directly. Its second premise accounts for the implicit dependency of abstract iterate cells $\underline{\ell}^{(i)}$ for $i > 0$ on the fixed point cell $\underline{\ell}$ (corresponding to the loop back edge in the control-flow graph); it suffices to check that the 1st abstract iterate cell has been emptied, as all abstract iterates $\underline{\ell}^{(i)}$ for $i > 1$ are reachable from $\underline{\ell}^{(1)}$. The third premise ensures that DAIG well-formedness is preserved, preventing emptying of initial abstract state or program statement references.

The E-Propagate rule recursively empties reference cells that depend on the edited cell, eventually bottoming out when no such cells are non-empty and E-Commit can be used to derive the $\mathcal{D}' \vdash n \Leftarrow v_\varepsilon \,;\, \mathcal{D}''$ premise. Note that there is no recomputation here in E-Propagate, only emptying.

The E-Loop rule applies when the final abstract iterate of a loop is dirtied. The sources of its fix edge are reset to its 0th and 1st abstract iterates and dirtying continues from the 1st abstract iterate. This handling is slightly more conservative than necessary in the case that an intermediate abstract iterate (i.e., for $i > 0$) is mutated externally, but it simplifies the presentation. This treament is precise if only statement cells can be edited, as all intermediate abstract iterate cells depend on any loop-body statement (transitively).

6 Soundness, Termination, and From-Scratch Consistency

In this section, we state two key properties of demanded abstract interpretation graphs: *from-scratch consistency*, which guarantees that DAIG query results are identical to the analysis results computed by the underlying abstract interpreter at a global fixed-point, and *query termination*, which guarantees termination of the DAIG query semantics even in the presence of unbounded abstract loop unrolling. The proofs are deferred to the appendix due to space constraints.

Both theorems rely on the preservation of DAIG well-formedness (Definition 4.1), DAIG–CFG consistency (Definition 4.2), and DAIG–AI consistency (Definition 4.3) under the operational semantics of Section 5.

Lemma 6.1 (DAIG Well-Formedness Preservation). *If $\mathcal{D}$ is well-formed and either $\mathcal{D}, M \vdash n \Rightarrow v \,;\, \mathcal{D}', M'$ or $\mathcal{D} \vdash n \Leftarrow v_\varepsilon \,;\, \mathcal{D}'$, then $\mathcal{D}'$ is well-formed.*

Lemma 6.2 (DAIG–CFG Consistency Preservation). *If $\mathcal{D} \cong \langle L, E, \ell_0\rangle$ and $\mathcal{D}, M \vdash n \Rightarrow v \,;\, \mathcal{D}', M'$ then $\mathcal{D}' \cong \langle L, E, \ell_0\rangle$.*

Lemma 6.3 (DAIG–AI Consistency Preservation). *If $\mathcal{D} \cong \langle \Sigma^\sharp, \varphi_0, [\![\cdot]\!]^\sharp, \sqsubseteq, \sqcup, \nabla\rangle$ and $\mathcal{D}, M \vdash n \Rightarrow v \,;\, \mathcal{D}', M'$ then $\mathcal{D}' \cong \langle \Sigma^\sharp, \varphi_0, [\![\cdot]\!]^\sharp, \sqsubseteq, \sqcup, \nabla\rangle$*

With these preservation results, we can now prove that DAIG query results for abstract states at program locations are from-scratch consistent with the global fixed-point invariant map of the DAIG's underlying abstract interpreter.

Theorem 6.1 (DAIG From-Scratch Consistency). *For all sound M and well-formed $\mathcal{D}$ such that $\mathcal{D} \cong \langle L, E, \ell_0\rangle$ and $\mathcal{D} \cong \langle \Sigma^\sharp, \varphi_0, [\![\cdot]\!]^\sharp, \sqsubseteq, \sqcup, \nabla\rangle$, if $\mathcal{D}, M \vdash \underline{\ell} \Rightarrow v \,;\, \mathcal{D}', M'$ then $v = [\![\ell]\!]^{\sharp *}_{\langle L, E, \ell_0\rangle}$.*

Corollary 6.2. *Query results are sound.*

Since the global invariant map $[\![\cdot]\!]^{\sharp *}$ of the underlying abstract interpreter $\langle \Sigma^\sharp, \varphi_0, [\![\cdot]\!]^\sharp, \sqsubseteq, \sqcup, \nabla\rangle$ is sound (by Global Abstract Interpreter Soundness (Proposition 3.2)) and a DAIG query for the abstract state at a location ℓ returns $[\![\ell]\!]^{\sharp *}$, DAIG query results themselves are sound.

Theorem 6.3 (DAIG Query Termination). *For all M and well formed $\mathcal{D}$ such that $\mathcal{D} \cong \langle L, E, \ell_0\rangle$ and $\mathcal{D} \cong \langle \Sigma^\sharp, \varphi_0, [\![\cdot]\!]^\sharp, \sqsubseteq, \sqcup, \nabla\rangle$, if n is in the namespace of $\mathcal{D}$ then there exist $v, \mathcal{D}', M'$ such that $\mathcal{D}, M \vdash n \Rightarrow v \,;\, \mathcal{D}', M'$.*

7 Implementation and Evaluation

Here, we describe a prototype implementation and evaluation of demanded abstract interpretation via our DAIG framework. Our evaluation studied two research questions:

- **Expressivity:** Does the DAIG framework allow for clean and straightforward implementations of rich analysis domains that cannot be handled by existing incremental and/or demand-driven frameworks?
- **Scalability:** For these rich analysis domains, what degree of performance improvement can be obtained by performing incremental and/or demand-driven analysis, as compared to batch analysis?

7.1 Implementation

Our DAIG framework is implemented in approximately 2,500 lines of OCaml code. Incremental and demand-driven analysis logic, including demanded unrolling, operates over an explicit graph representation of DAIGs, but per-function memoization (i.e., the auxiliary memo table M in the Fig. 8 semantics) is provided via `adapton.ocaml`, an open-source implementation of the technique of Hammer et al. [20].

Our implementation is parametric in an abstract domain, and the effort required to instantiate the framework to a new abstract domain is comparable to the effort required to do so in a classical abstract interpreter framework. The required module signature is essentially the abstract interpreter signature $\langle \Sigma^\sharp, \varphi_0, [\![\cdot]\!]^\sharp, \sqsubseteq, \sqcup, \nabla \rangle$, extended with some standard utilities which can often be automatically derived.

Interprocedurality. Although the formalism is defined over control-flow graphs for clarity and brevity, our implementation supports context-sensitive analysis of non-recursive programs with static calling semantics (i.e., no virtual dispatch or higher-order functions).

In order to analyze such programs, we initially construct a DAIG only for the "main" procedure in the initial context. Then, when a query is issued for the abstract state after a call, we construct a DAIG for its callee in the proper context. When a query is issued at a location/context for which no DAIG has yet been constructed, we construct the DAIG for its containing function and analyze dataflow to its entry.

These operations are parametric in a context-sensitivity policy for choosing a context in which to analyze a callee at a call site. Our implementation includes functors that implement context-insensitivity and also 1- and 2-call-site-sensitivity [37].

7.2 Expressivity

To demonstrate the expressivity of our DAIG framework, we have instantiated it with three existing well-known abstract interpretation techniques — interval, octagon and shape analysis — all of which are inexpressible using existing incremental and/or demand-driven analysis frameworks. Here, we describe our experience applying the DAIG-based interval and shape analyses to a small set of programs. In Section 7.3, we use the octagon domain to investigate the scalability of demanded analysis on synthetic benchmarks.

Together, these analysis implementations provide evidence for our approach's agnosticity to the underlying abstract domain, including domains with black-box external dependencies and/or complicated non-monotone abstract operations.

Interval Analysis. The interval abstract domain is a textbook example of an infinite-height lattice, requiring widening to guarantee analysis convergence. An interval $[l, u]$ abstracts the set of numbers between lower bound l and upper bound u. Join and widen operations and abstract states are defined in the standard way [11]. Abstract interpretation in this domain is known as interval analysis and is commonly used, e.g., to verify the safety of array accesses. Interval analysis has been applied at industrial scale, for example by Cousot et al. [12].

In practice, it is common to use an optimized off-the-shelf interval abstract domain such as that of APRON [24] or Elina [38]. We have implemented an APRON-backed interval analysis for JavaScript programs in the DAIG framework. As an indication of the flexibity of our framework, we were able to use the APRON library *without modification.*

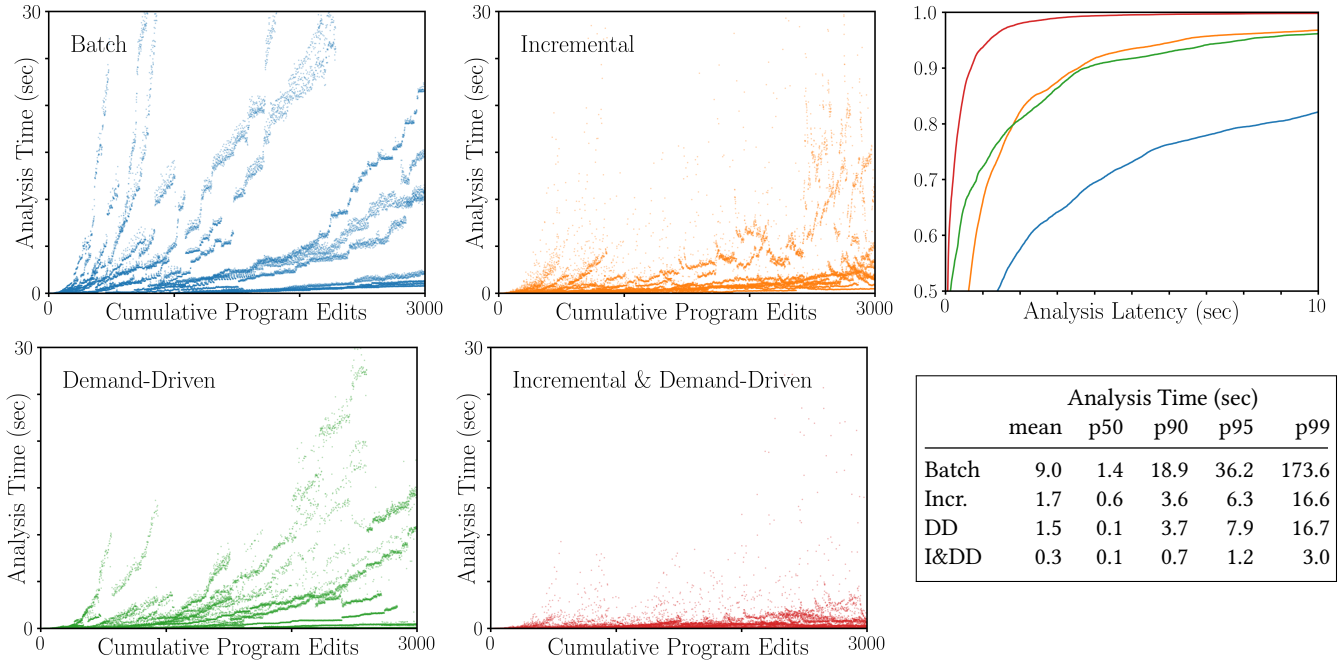
In order to validate our implementation, we analyzed 23 array-manipulating programs — with functions such as `contains`, `equals`, `swap`, and `indexof` — from the test suite of Buckets.JS, a JavaScript data structure library [36].

Using the 2-call-string-sensitive context policy, our analysis verified the safety of all 85 array accesses in the programs; with 1-call-string-sensitivity, it verified 71/74 (96%), and with context-insensitive analysis it verified 4/18 (22%). These figures show that standard numerical analyses on DAIGs behave as they would in a batch analysis engine.

Shape Analysis. Precise analysis of recursive data structures such as linked lists is essential in many domains. Such analysis relies on complex abstract domains that cannot be expressed in existing frameworks for incremental and demand-driven analysis. We have implemented a DAIG-based demanded shape analysis for singly-linked lists. An abstract state in this shape domain is a triple consisting of

- A separation logic formula over points-to ($\alpha.f \mapsto \alpha'$) and list-segment ($\texttt{lseg}(\alpha, \alpha')$) atomic propositions, stating respectively that the f field of the object at symbolic address α points to α' and that there exists a sequence of `next` pointer dereferences from α to α' [32],
- A collection of pure constraints: equalities and disequalities over memory addresses, and
- An environment mapping variables to memory addresses.

Join, widen, and implication all rely on a collection of rewrite rules over such states from Chang et al. [10] (specialized to a fixed inductive definition for list segments). All told, the implementation of this shape domain requires approximately 500 lines of OCaml code.



	Analysis Time (sec)				
	mean	p50	p90	p95	p99
Batch	9.0	1.4	18.9	36.2	173.6
Incr.	1.7	0.6	3.6	6.3	16.6
DD	1.5	0.1	3.7	7.9	16.7
I&DD	0.3	0.1	0.7	1.2	3.0

Figure 10. Performance of octagon analysis on the synthetic workload of interleaved program edits and analysis queries described in Section 7.3. The four scatter plots show the scaling of each configuration as the program size is increased by edits, and their color-coding serves as a legend to the fifth figure: a cumulative distribution plot showing the fraction of analysis runs (y axis) completed by each configuration within some time interval (x axis). Lastly, the table shows summary statistics for each configuration, including the mean, median, 90th, 95th, and 99th percentile analysis latency.

We have applied this DAIG-based shape analysis to successfully verify the correctness and memory-safety of the list `append` procedure of Fig. 2, along with several linked list utilities from the aforementioned Buckets.js library including `foreach` and `indexof` [36]. Analysis of the ℓ_3-to-ℓ_4-to-ℓ_3 loop of the list `append` procedure converges in one demanded unrolling with a precise result.

7.3 Scalability

To study scalability, we conducted an initial investigation of what performance improvements are possible with demanded analysis variants in our framework. We compared the performance of analysis with and without incrementality and demand on interleaved sequences of program edits and queries. Throughout this section, our framework is instantiated with a context-insensitive APRON-backed octagon domain: a relational numerical domain representing invariants of the form $\pm x \pm y \leq y$, widely used in practice due to its balance of expressivity and efficiency [27].

To exercise the analyses, we created synthetic workloads consisting of 3,000 random edits to an initially-empty program. Programs are generated in a JavaScript subset with assignment, arrays, conditional branching, while loops, and (non-recursive) function calls of the form `x = f(y)`. An "edit" is an insertion of a randomly generated statement, if-then-else conditional, or while loop at a randomly-sampled program location, with 85%, 10%, and 5% probability respectively, and statements and expressions are generated probabilistically from their respective grammars.

We evaluate four analysis configurations on this workload:

(1) *Batch* analysis: Classical whole-program abstract interpretation, fully re-analyzing the entire program from scratch in response to each edit.
(2) *Incremental* analysis: An incremental-only configuration which applies the edit semantics to dirty as few previously-computed analysis results as possible, but eagerly recomputes all dirtied cells.
(3) *Demand-driven* analysis: A demand-driven-only configuration which dirties the full DAIG after each edit, but applies the query semantics to avoid computing analysis results that aren't demanded.
(4) *Incremental & demand-driven* analysis: The full demanded abstract interpretation technique, which applies both the edit and query semantics to maximize reuse and minimize redundant computation.

In the demand-driven configurations, queries are issued at five randomly-sampled program locations between each edit. Note that since the first three configurations were implemented atop our DAIG framework, which is designed to

support both incremental and demand-driven analysis, they may not be as tuned as specialized implementations.

Each plot includes data points from 9 separate trials, with fixed random seeds such that the same edits (and, in the two demand-driven configurations, queries) are issued to each configuration. In total, this data set includes 27,000 analysis executions in each exhaustive configuration and 135,000 queries in each demand-driven configuration.

The results, as shown in Fig. 10, indicate that while incremental and demand-driven analysis each significantly improve analysis latencies with respect to the batch analysis baseline, combining the two provides an additional large reduction in latency. This effect is most apparent in the tail of the distribution, since edits that dirty large regions of the program are costly for incremental analysis, and queries that depend on large regions of the graph are costly for demand-driven analysis. By combining incremental dirtying with demand-driven evaluation, demanded abstract interpretation mitigates these worst-case scenarios and consistently keeps analysis costs low even as the program grows.

In particular, at the 95^{th} percentile, the 1.2s latency of incremental demand-driven analysis is more than five times lower than the next best configuration, and potentially low enough to support interactive use. Fig. 10 gives a cumulative distribution of analysis latencies, again showing the large advantage of the incremental demand-driven analysis over other configurations.

8 Related Work

Incremental Computation. Techniques for the efficient caching and reuse of computation results, particularly those based on memoization of pure functions [1, 19, 28] and dependency graphs [13, 30], have been the subject of a great deal of research and seen widespread practical application.

More recently, dependency graph-based approaches to incremental computation have improved on generic memoization and graph-based techniques, allowing for fine-grained automatic caching and reuse even in the presence of changes to inputs or an underlying data store [2, 3]. Building on these graph-based techniques for self-adjusting computation, some recent work has focused on support for interactive and demand-driven computations [20, 21]. Although this approach yields a general and powerful system for incremental computation, its low-level primitives make it difficult to express the complex fixed-point computation over cyclic control-flow graphs in arbitrary abstract interpretations. We take inspiration from demanded computation graphs but instead specialize the language of demanded computations to demanded abstract interpretations, both with syntactic structures and with a query/edit semantics which dynamically modifies the dependency graph to model such computations.

Incremental Analysis. The application of incremental computation to program analysis is similarly well-studied, going back at least to the development of incremental dataflow analyses to support responsive continuous compilation [33, 43]. Recent work has contributed incremental versions of several classes of program analysis, including IFDS/IDE dataflow analyses [5, 16] and analyses based on extensions to Datalog [41, 42]. These specialized approaches offer effective solutions for certain classes of program analysis, but place restrictions on abstract domains that rule out arbitrary abstract interpretations in infinite-height domains.

Compositional program analysis, in which summaries are computed for individual files or compilation units rather than a whole program, naturally supports incrementality in the sense that results need only be recomputed for changed files. This has shown to be very effective for scaling program analyses to massive codebases in CI/CD systems [9, 14, 18], but it operates at a much coarser granularity than both the aforementioned approaches and our own, since it is designed to scale up to massive programs rather than to minimize analysis latencies at development-time.

Leino and Wüstholz [25] propose an fine-grained incremental verification technique for the Boogie language, which verifies user-provided specifications of imperative procedures. These specifications include loop invariants, allowing their algorithm to ignore cyclic dependencies altogether.

Demand-Driven Analysis. Demand-driven techniques for dataflow analysis are also well-studied. The intra-procedural problem was studied by Babich and Jazayeri [6]. Several extensions to inter-procedural analysis have been presented, for example, by Reps [31], Duesterwald et al. [17], and Sagiv et al. [35]. In nearly all cases previous work has been focused on finite domains. The work of Sagiv et al. [35] allows for infinite domains of finite height, but does not consider infinite-height domains like intervals.

Any static analysis expressible as a context-free-language reachability (CFL-reachability) problem can be computed in a demand-driven fashion as a "single-source" problem [29]. As such, a number of papers have presented demand-driven algorithms for flow-insensitive pointer analysis [22, 39, 40].

9 Conclusion

We have presented a novel framework for demanded abstract interpretation, in which an arbitrary abstract interpretation can be made both incremental and demand-driven. Unlike previous frameworks, ours supports arbitrary lattices and widening operators. The framework is based on a novel demanded abstract interpretation graph (DAIG) representation of the analysis problem, where careful handling of loops ensures the DAIG remains acyclic. We have proved various key properties of the framework, including soundness, termination, and from-scratch consistency. Our implementation shows that complex analyses can be easily implemented with our framework, with the potential for significant performance wins in incremental and demand-driven scenarios.

References

[1] Martín Abadi, Butler W. Lampson, and Jean-Jacques Lévy. 1996. Analysis and Caching of Dependencies. In *ICFP*. ACM, 83–91.

[2] Umut A. Acar, Amal Ahmed, and Matthias Blume. 2008. Imperative self-adjusting computation. In *POPL*. ACM, 309–322.

[3] Umut A. Acar, Guy E. Blelloch, and Robert Harper. 2002. Adaptive functional programming. In *POPL*. ACM, 247–259.

[4] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA.

[5] Steven Arzt and Eric Bodden. 2014. Reviser: efficiently updating IDE-/IFDS-based data-flow analyses in response to incremental program changes. In *36th International Conference on Software Engineering, ICSE '14*, Pankaj Jalote, Lionel C. Briand, and André van der Hoek (Eds.). ACM, 288–298.

[6] Wayne A. Babich and Mehdi Jazayeri. 1978. The Method of Attributes for Data Flow Analysis: Part II. Demand analysis. *Acta Informatica* 10, 3 (1978), 265–272.

[7] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 2005. Smallfoot: Modular Automatic Assertion Checking with Separation Logic. In *FMCO (Lecture Notes in Computer Science)*, Vol. 4111. Springer, 115–137.

[8] François Bourdoncle. 1993. Efficient chaotic iteration strategies with widenings. In *Formal Methods in Programming and Their Applications (Lecture Notes in Computer Science)*, Vol. 735. Springer, 128–141.

[9] Cristiano Calcagno and Dino Distefano. 2011. Infer: An Automatic Program Verifier for Memory Safety of C Programs. In *NASA Formal Methods (Lecture Notes in Computer Science)*, Vol. 6617. Springer, 459–465.

[10] Bor-Yuh Evan Chang, Xavier Rival, and George C. Necula. 2007. Shape Analysis with Structural Invariant Checkers. In *SAS (Lecture Notes in Computer Science)*, Vol. 4634. Springer, 384–401.

[11] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages,1977*, Robert M. Graham, Michael A. Harrison, and Ravi Sethi (Eds.). ACM, 238–252.

[12] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2005. The ASTREÉ Analyzer. In *ESOP (Lecture Notes in Computer Science)*, Vol. 3444. Springer, 21–30.

[13] Alan J. Demers, Thomas W. Reps, and Tim Teitelbaum. 1981. Incremental Evaluation for Attribute Grammars with Application to Syntax-Directed Editors. In *POPL*. ACM Press, 105–116.

[14] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. 2019. Scaling static analyses at Facebook. *Commun. ACM* 62, 8 (2019), 62–70.

[15] Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. 2006. A Local Shape Analysis Based on Separation Logic. In *Tools and Algorithms for the Construction and Analysis of Systems, 12th International Conference, TACAS 2006 (Lecture Notes in Computer Science)*, Holger Hermanns and Jens Palsberg (Eds.), Vol. 3920. Springer, 287–302.

[16] Lisa Nguyen Quang Do, Karim Ali, Benjamin Livshits, Eric Bodden, Justin Smith, and Emerson R. Murphy-Hill. 2017. Just-in-time Static Analysis. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*, Tevfik Bultan and Koushik Sen (Eds.). ACM, 307–317. https://doi.org/10.1145/3092703.3092705

[17] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. 1995. Demand-Driven Computation of Interprocedural Data Flow. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*.

[18] Manuel Fähndrich and Francesco Logozzo. 2010. Static Contract Checking with Abstract Interpretation. In *FoVeOOS (Lecture Notes in Computer Science)*, Vol. 6528. Springer, 10–30.

[19] John Field and Tim Teitelbaum. 1990. Incremental Reduction in the lambda Calculus. In *LISP and Functional Programming*. ACM, 307–322.

[20] Matthew A. Hammer, Joshua Dunfield, Kyle Headley, Nicholas Labich, Jeffrey S. Foster, Michael W. Hicks, and David Van Horn. 2015. Incremental computation with names. In *OOPSLA*. ACM, 748–766.

[21] Matthew A. Hammer, Yit Phang Khoo, Michael Hicks, and Jeffrey S. Foster. 2014. Adapton: composable, demand-driven incremental computation. In *PLDI*. ACM, 156–166.

[22] Nevin Heintze and Olivier Tardieu. 2001. Demand-Driven Pointer Analysis. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[23] Susan Horwitz, Thomas W. Reps, and Shmuel Sagiv. 1995. Demand Interprocedural Dataflow Analysis. In *SIGSOFT FSE*. ACM, 104–115.

[24] Bertrand Jeannet and Antoine Miné. 2009. Apron: A Library of Numerical Abstract Domains for Static Analysis. In *CAV (Lecture Notes in Computer Science)*, Vol. 5643. Springer, 661–667.

[25] K. Rustan M. Leino and Valentin Wüstholz. 2015. Fine-Grained Caching of Verification Results. In *CAV (1) (Lecture Notes in Computer Science)*, Vol. 9206. Springer, 380–397.

[26] Stephen Magill, Aleksandar Nanevski, Edmund Clarke, and Peter Lee. 2006. Inferring invariants in separation logic for imperative list-processing programs. In *SPACE*. Citeseer.

[27] Antoine Miné. 2006. The octagon abstract domain. *High. Order Symb. Comput.* 19, 1 (2006), 31–100.

[28] William Pugh and Tim Teitelbaum. 1989. Incremental Computation via Function Caching. In *POPL*. ACM Press, 315–328.

[29] Thomas Reps. 1998. Program analysis via graph reachability. *Information and Software Technology* 40, 11-12 (November/December 1998), 701–726.

[30] Thomas W. Reps. 1982. Optimal-Time Incremental Semantic Analysis for Syntax-Directed Editors. In *POPL*. ACM Press, 169–176.

[31] Thomas W. Reps. 1994. Solving Demand Versions of Interprocedural Analysis Problems. In *CC*.

[32] John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002)*. IEEE Computer Society, 55–74.

[33] Barbara G. Ryder. 1983. Incremental Data Flow Analysis. In *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages*. ACM Press, 167–176.

[34] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. 2018. Lessons from building static analysis tools at Google. *Commun. ACM* 61, 4 (2018), 58–66.

[35] Shmuel Sagiv, Thomas W. Reps, and Susan Horwitz. 1996. Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation. *Theor. Comput. Sci.* 167, 1&2 (1996), 131–170.

[36] Mauricio Santos. 2016. *Buckets-JS: A JavaScript Data Structure Library*. https://github.com/mauriciosantos/Buckets-JS.

[37] Micha Sharir and Amir Pnueli. 1981. Two Approaches to Interprocedural Data Flow Analysis. In *Program Flow Analysis: Theory and Applications*. Prentice-Hall.

[38] Gagandeep Singh, Markus Püschel, and Martin T. Vechev. 2017. Fast polyhedra abstract domain. In *POPL*. ACM, 46–59.

[39] Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. 2016. Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java. In *30th European Conference on Object-Oriented Programming (ECOOP)*.

[40] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. 2005. Demand-Driven Points-To Analysis for Java. In *OOPSLA*.

[41] Tamás Szabó, Gábor Bergmann, Sebastian Erdweg, and Markus Voelter. 2018. Incrementalizing lattice-based program analyses in Datalog. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 139:1–139:29.

[42] Tamás Szabó, Sebastian Erdweg, and Markus Voelter. 2016. IncA: a DSL for the definition of incremental program analyses. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016*, David Lo, Sven Apel, and Sarfraz Khurshid (Eds.). ACM, 320–331.

[43] F. Kenneth Zadeck. 1984. Incremental data flow analysis in a structured program editor. In *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction, Montreal, Canada, June 17-22, 1984*. ACM, 132–143.

A Proofs for Section 4 (Demanded AI Graphs)

First, we fix some notation and terminology related to flow graphs and their structure, all of which is fairly standard:

The edges E of a reducible flow graph program $\langle L, E, \ell_0 \rangle$ may be partitioned disjointly into a set E_f of *forward* edges and a set E_b of *back* edges, where forward edges form a directed acyclic graph that spans L and the destination ℓ' of each back edge $\ell \dashv[s]\mapsto \ell' \in E_b$ dominates[6] the source ℓ. A vertex is a *loop head* if it is the destination of an edge in E_b.

Furthermore, each back edge $e = \ell \dashv[s]\mapsto \ell' \in E_b$ in a reducible flow graph uniquely determines a *natural loop* $\text{loop}(\ell') : \mathcal{P}(L)$, the set of locations from which ℓ can be reached without passing through ℓ'. Intuitively, the natural loop is the "body" of the loop with head ℓ', and we define $\text{loop}(\ell) \triangleq \emptyset$ for all non-loop head locations ℓ.

There are well-known efficient algorithms both to partition forward/back edges and to find natural loops in reducible flow graphs (e.g., [4], Sec. 9.6) which we do not repeat here.

We also define a helper function $\text{fwd-edges-to}_{\langle L,E,\ell_0\rangle} : Loc \to \mathcal{P}(\mathbb{N} \times Edge)$ which assigns indices to forward control-flow edges into the given location. These indices will be essential for DAIG construction, allowing us to uniquely name the intermediate abstract states before a join.

We denote by $L_\sqcup$ the set of CFG join points $\left\{\ell \;\middle|\; |\text{fwd-edges-to}(\ell)| \geq 2\right\}$ and by $L_{\not\sqcup}$ its complement $L / L_\sqcup$. Note that our definition is non-standard: these points are determined by *forwards* indegree rather than *total* indegree, since no join operation is necessary at a loop entry with only a single non-loop predecessor.

Definition A.1 (DAIG–CFG Consistency)**.** A DAIG $\mathcal{D} = \langle R, C \rangle$ is consistent with a program CFG $\langle L, E, \ell_0 \rangle$, written $\mathcal{D} \cong \langle L, E, \ell_0 \rangle$, when it is well-formed and its structure accurately encodes the abstract interpretation of that program by satisfying the following conditions. See Fig. 7 for a visual representation of this definition to build intuition for the more formal statements below.

(1) Forward CFG edges to *non-join* locations are encoded by a transfer function edge connecting the abstract pre-state and statement to the abstract post-state.

$$\forall \ell' \in L_{\not\sqcup},\ \ell \dashv[s]\mapsto \ell' \in E_f \quad . \quad \underline{\ell} \cdot \underline{\ell'}[s : Stmt] \in R \quad \wedge \quad n_{\ell'} \leftarrow [\![\cdot]\!]^\sharp(n_\ell \cdot n_{\ell'}, n_\ell) \in C$$

(2) Forward CFG edges to *join* locations are slightly more complex: for each join location ℓ, fwd-edges-to assigns a unique integer index i to each incoming CFG edge. For each such CFG edge, a transfer function edge connects the reference cells containing its statement label and the abstract state at its source location to a pre-join abstract state (named by $\underline{i} \cdot n_\ell$) specific to that edge. Then, a single join edge connects each pre-join abstract state $\underline{i} \cdot n_\ell$ to n_ℓ, the actual abstract state at ℓ.

$$\forall \ell \in L_\sqcup \;.\quad n_\ell \leftarrow \sqcup(\underline{1} \cdot n_\ell, \ldots, \underline{|\text{fwd-edges-to}(\ell)|} \cdot n_\ell) \in C \quad \wedge \quad \left(\begin{array}{c} \forall (i, \ell_i \dashv[s_i]\mapsto \ell) \in \text{fwd-edges-to}(\ell)\;. \\ \underline{i} \cdot \underline{\ell_i} \cdot \underline{\ell}[s_i : Stmt] \in R \;\wedge\; \underline{i} \cdot n_\ell \leftarrow [\![\cdot]\!]^\sharp(\underline{i} \cdot \underline{\ell_i} \cdot \underline{\ell},\ n_{\ell'}) \in C \end{array} \right)$$

(3) CFG back edges are encoded by some k analysis iterations over the corresponding loop body. First, a transfer function edge connects the reference cell (named by $n_{\ell'} = \underline{\ell'}^{(0)}$) at the end of the first loop iteration and the back edge's statement label to the first pre-widen abstract state at the loop head (named by $\underline{\ell}^{(0)} \cdot \underline{\ell}^{(1)}$). Then, for each $i \in [0, k)$, the loop body is unrolled with iteration count i, and widen edges connect the ith abstract iterate and pre-widen abstract state to the next abstract iterate. Lastly, a fix edge connects the final two abstract iterates to the loop head fixed point $\underline{\ell}$.

$$\forall \ell' \dashv[s]\mapsto \ell \in E_b \;.\; \exists\, k \geq 1 \;. \begin{array}{l} \underline{\ell'} \cdot \underline{\ell}[s : Stmt] \in R \;\wedge \\ \underline{\ell}^{(0)} \cdot \underline{\ell}^{(1)} \leftarrow [\![\cdot]\!]^\sharp(\underline{\ell'} \cdot \underline{\ell}, \underline{\ell'}^{(0)}) \in C \;\wedge \\ \underline{\ell} \leftarrow \text{fix}(\underline{\ell}^{(k-1)}, \underline{\ell}^{(k)}) \in C \;\wedge \\ \forall i \in [0, k)\;. \left(\begin{array}{l} \underline{\ell}^{(i)} \leftarrow \nabla(\underline{\ell}^{(i-1)},\ \underline{\ell}^{(i-1)} \cdot \underline{\ell}^{(i)}) \in C \;\wedge \\ \left\{ \text{incr-c}^i(c) \;\middle|\; c \in \text{loop-comps}(\ell) \right\} \subseteq C \end{array} \right) \end{array}$$

The initial name n_ℓ for a location ℓ is $\underline{\ell}^{(0)}$ if it belongs to any natural loop and $\underline{\ell}$ if it does not, and we denote by $\text{loop-comps}(\ell)$ the subset of C whose elements contain a name $\underline{\ell'}^{(0)}$ such that $\ell' \in \text{loop}(\ell)$. The helper function incr increments the loop-iteration counts in names, and we write incr-c for the pointwise lifting of incr to edges $c \in Comp$.

$$\text{incr}(n) \quad \triangleq \begin{cases} \underline{\ell}^{(k+1)} & \text{if } n = \underline{\ell}^{(k)} \\ \text{incr}(n_1) \cdot \text{incr}(n_2) & \text{if } n = n_1 \cdot n_2 \\ n & \text{otherwise} \end{cases}$$

[6] A vertex ℓ *dominates* a vertex ℓ' if every path from the entry to ℓ' passes through ℓ.

Definition A.2 (Initial DAIG Construction). The initial DAIG $\mathcal{D}_{\text{init}}(\langle L, E, \ell_0\rangle, \langle \Sigma^\sharp, \varphi_0, [\![\cdot]\!]^\sharp, \sqsubseteq, \sqcup, \nabla\rangle)$ for program $\langle L, E, \ell_0\rangle$ and abstract domain $\langle \Sigma^\sharp, \varphi_0, [\![\cdot]\!]^\sharp, \sqsubseteq, \sqcup, \nabla\rangle$. consists of several distinct categories of both reference cells and computation edges, which we describe and define separately for clarity and ease of presentation.

Reference cells in R_{Stmt} contain program statements, those in $R_{\Sigma^\sharp}$ contain abstract states at each control location and join point, and those in $R_{\circlearrowleft}$ contain those needed to encode loops, while computation edges in each C_f encode the uses of that function f. As in Definition 4.2, we denote by n_ℓ the initial name at ℓ, which is defined as $\underline{\ell}^{(0)}$ if ℓ is in the natural loop of some back edge in E_b, or $\underline{\ell}$ otherwise.

That is, $\mathcal{D}_{\text{init}}\left(\langle L, E, \ell_0\rangle, \langle \Sigma^\sharp, \varphi_0, [\![\cdot]\!]^\sharp, \sqsubseteq, \sqcup, \nabla\rangle\right) \triangleq \left\langle R_{Stmt} \cup R_{\Sigma^\sharp} \cup R_{\circlearrowleft},\ C_{[\![\cdot]\!]^\sharp} \cup C_\sqcup \cup C_{\text{fix}} \cup C_\nabla \right\rangle$, where:

- Reference cells in R_{Stmt} each contain a program statement and are named by the locations before and after that statement, along with a disambiguating integer index when multiple forwards[7] control-flow edges share a destination.

$$R_{Stmt} \triangleq \left\{\underline{\ell}\cdot\underline{\ell'}[s : Stmt] \,\middle|\, (\ell' \in L_{\not\downarrow} \ \wedge\ \ell \text{-}[s]\!\!\rightarrow \ell' \in E) \vee \quad \ell \text{-}[s]\!\!\rightarrow \ell' \in E_b\right\}$$
$$\cup \left\{\underline{i}\cdot\underline{\ell}\cdot\underline{\ell'}[s : Stmt] \,\middle|\, \ell \in L_\sqcup \ \wedge\ (i, \ell' \text{-}[s]\!\!\rightarrow \ell) \in \text{fwd-edges-to}(\ell)\right\}$$

- $R_{\Sigma^\sharp}$ contains one reference cell of abstract state type per program location, with the initial location ℓ_0's cell populated by the initial abstract state φ_0, along with indexed reference cells for the pre-join abstract states at join point locations.

$$R_{\Sigma^\sharp} \triangleq \left\{\underline{\ell_0}[\varphi_0 : \Sigma^\sharp]\right\} \cup \left\{n_\ell[\varepsilon : \Sigma^\sharp] \,\middle|\, \ell \in L/\{\ell_0\}\right\} \cup \left\{\underline{i}\cdot n_\ell[\varepsilon : \Sigma^\sharp] \,\middle|\, \ell \in L_\sqcup \ \wedge\ 1 \le i \le |\text{fwd-edges-to}(\ell)|\right\}$$

- Reference cells in $R_{\circlearrowleft}$ encode the zeroth ($\underline{\ell}^{(0)}$) and first ($\underline{\ell}^{(1)}$) abstract iterates and the first pre-widening abstract state ($\underline{\ell}^{(0)}\cdot\ \underline{\ell}^{(1)}$) at each loop head ℓ. Names for further iterations' intermediate abstract states are dynamically generated as needed by the operational semantics via demanded unrolling.

$$R_{\circlearrowleft} \triangleq \left\{\underline{\ell}^{(i)}[\varepsilon : \Sigma^\sharp] \,\middle|\, \ell' \text{-}[s]\!\!\rightarrow \ell \in E_b \ \wedge\ i \in \{0, 1\}\right\} \cup \left\{\underline{\ell}^{(0)}\cdot\ \underline{\ell}^{(1)}[\varepsilon : \Sigma^\sharp] \,\middle|\, \ell' \text{-}[s]\!\!\rightarrow \ell \in E_b\right\}$$

- Computations in $C_{[\![\cdot]\!]^\sharp}$ encode abstract transfers. Its three subsets respectively encode the abstract semantics of forward edges to non-join locations, forward edges to join locations, and back edges.
We define the shorthand src-nm(ℓ, ℓ') to be $\underline{\ell}$ when ℓ is a loop head and $\ell' \notin \text{loop}(\ell)$ and n_ℓ otherwise.

$$C_{[\![\cdot]\!]^\sharp} \triangleq \left\{n_\ell \leftarrow [\![\cdot]\!]^\sharp\left(\underline{\ell'}\cdot\underline{\ell}, \text{src-nm}(\ell', \ell)\right) \,\middle|\, \ell \in L_{\not\downarrow} \ \wedge\ \ell' \text{-}[s]\!\!\rightarrow \ell \in E_f\right\}$$
$$\cup \left\{\underline{i}\cdot n_\ell \leftarrow [\![\cdot]\!]^\sharp\left(\underline{i}\cdot\underline{\ell'}\cdot\underline{\ell},\ \text{src-nm}(\ell', \ell)\right) \,\middle|\, \ell \in L_\sqcup \ \wedge\ (i,\ \ell' \text{-}[s]\!\!\rightarrow \ell) \in \text{fwd-edges-to}(\ell)\right\}$$
$$\cup \left\{\underline{\ell}^{(0)}\cdot\ \underline{\ell}^{(1)} \leftarrow [\![\cdot]\!]^\sharp(\underline{\ell'}\cdot\underline{\ell}, n_{\ell'}) \,\middle|\, \ell' \text{-}[s]\!\!\rightarrow \ell \in E_b\right\}$$

- Computations in $C_\sqcup$ encode joins at program locations of forwards indegree ≥ 2; the abstract state at ℓ is the least upper bound of the abstract states on each incoming edge to ℓ.

$$C_\sqcup \triangleq \left\{n_\ell \leftarrow \sqcup(\underline{1}\cdot n_\ell, \ldots, \underline{k}\cdot n_\ell) \,\middle|\, \ell \in L_\sqcup \ \wedge\ k = |\text{fwd-edges-to}(\ell)|\right\}$$

- Computations in C_{fix} connect the 0th and 1st abstract iterates at a loop head to its fixed point. The symbol fix is not a function *per se* but rather an indicator of a CFG back edge that is abstractly unrolled by the operational semantics without introducing cyclic dependencies in the static DAIG structure itself.

$$C_{\text{fix}} \triangleq \left\{\underline{\ell} \leftarrow \text{fix}\left(\underline{\ell}^{(0)}, \underline{\ell}^{(1)}\right) \,\middle|\, \ell' \text{-}[s]\!\!\rightarrow \ell \in E_b\right\}$$

- Finally, C_∇ contains widening computations at loop heads; in its initial state, the DAIG includes only the first widen for each loop.

$$C_\nabla \triangleq \left\{\underline{\ell}^{(1)} \leftarrow \nabla(\underline{\ell}^{(0)},\ \underline{\ell}^{(0)}\cdot\ \underline{\ell}^{(1)}) \,\middle|\, \ell' \text{-}[s]\!\!\rightarrow \ell \in E_b\right\}$$

Lemma A.1 (Initial DAIG Well-Formedness, CFG-Consistency, and AI-Consistency). *For all well-formed programs* $\langle L, E, \ell_0\rangle$, $\mathcal{D}_{init}(\langle L, E, \ell_0\rangle)$ *is well-formed,* $\mathcal{D}_{init}(\langle L, E, \ell_0\rangle) \cong \langle L, E, \ell_0\rangle$, *and* $\mathcal{D}_{init}(\langle L, E, \ell_0\rangle) \cong \langle \Sigma^\sharp, \varphi_0, [\![\cdot]\!]^\sharp, \sqsubseteq, \sqcup, \nabla\rangle$.

Proof. DAIG well-formedness holds by construction:

(1) Each component of R constructs reference cell names of a different structure, so no two reference names are equal.

[7] A reducible CFG has at most one back edge to each vertex, so no disambiguation is necessary for back edges.

(2) Each component of C destination names of a different structure, with the possible exception of $C_{\sqcup}$, C_{fix}, and the first component of $C_{[\![\cdot]\!]^\sharp}$, all of which may construct destination names of the form ℓ or $\ell^{(0)}$. However, since $L_{\sqcup}$ and $L_{\not\sqcup}$ are disjoint, there is no overlap between the destination names of $C_{\sqcup}$ and $C_{[\![\cdot]\!]^\sharp}$. Furthermore, there is no overlap between C_{fix} and either $C_{\sqcup}$ or $C_{[\![\cdot]\!]^\sharp}$ because if $\ell' \dashv[s]\mapsto \ell \in E_b$ then $n_\ell = \ell^{(0)} \neq \underline{\ell}$.
(3) By construction, $n_\ell \rightsquigarrow n_{\ell'}$ only if there exists a path in E_f from ℓ to ℓ'. CFG back edges $\ell' \dashv[s]\mapsto \ell$ in E_b maintain this antisymmetry; $n_{\ell'} \rightsquigarrow \underline{\ell}^{(0)} \cdot \underline{\ell}^{(1)} \rightsquigarrow \underline{\ell}^{(1)} \rightsquigarrow \underline{\ell}$ through the computations of $C_{[\![\cdot]\!]^\sharp}$, C_∇, and C_{fix} respectively, and $\underline{\ell} \not\rightsquigarrow n_{\ell'}$ since ℓ' is in the natural loop of ℓ and $\underline{\ell}$ may only appear as the source of edges *not* into ℓ's natural loop.
(4) Each edge in C is well-typed by construction; names of the form $\underline{\ell} \cdot \underline{\ell'}$ and and $\underline{i} \cdot \underline{\ell} \cdot \underline{\ell'}$ are of type *Stmt* and all others are of type $\Sigma^\sharp$.
(5) The name of each empty reference in R appears as the destination of an edge in C.

We elide the details of CFG-consistency, but it is straightforward to verify by cross-referencing the conditions of Definition 4.2 with the set constructions of Definition A.2. AI-consistency holds vacuously, as the only non-empty reference of type $\Sigma^\sharp$ in $\mathcal{D}_{\mathsf{init}}$ is $\underline{\ell_0}$, which contains φ_0. □

B Proofs for Section 6 (Soundness, Termination, and From-Scratch Consistency)

Lemma B.1 (DAIG Well-Formedness Preservation). *If $\mathcal{D}$ is well-formed and either $\mathcal{D}, M \vdash n \Rightarrow v\,;\mathcal{D}', M'$ or $\mathcal{D} \vdash n \Leftarrow v_\varepsilon\,;\mathcal{D}'$, then $\mathcal{D}'$ is well-formed.*

Proof. By structural induction on the operational semantics derivation.

Note that the only DAIG well-formedness condition that refers to reference cells contents is (5), which is trivially preserved by each query rule Q-$*$ since they never empty reference cells. Edit rule E-Commit's third premise ensures that edits don't violate (5), while E-Loop and E-Propagate only affect cells with non-zero indegree and therefore also preserve (5). Thus, only those rules that affect the *structure* of the DAIG must be considered: Q-Loop-Unroll and E-Loop.

Case Q-Loop-Unroll: By induction, $\mathcal{D}'$ and $\mathcal{D}''$ are well-formed. The unroll procedure preserves well-formedness condition (1) since $incr(n) \neq n$ for each n in the set of added reference cells $\left\{\mathsf{incr}(n)[\varepsilon : \Sigma^\sharp] \;\middle|\; \underline{\ell}^{(k-1)} \rightsquigarrow n \rightsquigarrow \underline{\ell}^{(k)}\right\}$ as each such n is an abstract state in a loop body and therefore has an iteration count; condition (2) since c is removed from C before it is replaced and by the same argument as (1) for the destination of each edge in $\left\{\mathsf{incr\text{-}c}(c) \;\middle|\; \underline{\ell}^{(k-1)} \rightsquigarrow \mathsf{dest}(c) \rightsquigarrow \underline{\ell}^{(k)}\right\}$; conditions (3) and (4) by isomorphism from the previous iteration's edges to the unrolled iteration's edges; and condition (5) since each added reference has a corresponding added incoming edge.

Therefore, $\mathsf{unroll}(\mathcal{D}'', c)$ is well-formed, so by induction $\mathcal{D}'''$ is well-formed.

Case E-Loop: Except for the fix edge whose inputs are changed, the intermediate DAIG $\langle R, C'\rangle$ is identical to $\mathcal{D}$, which is well-formed. Therefore, $\langle R, C'\rangle$ trivially satisfies well-formedness conditions (1) and (3), which refer only to references. It satisfies conditions (2) and (5) since each reference in R has the same number of incoming edges in C and C' because the added and removed edge have the same destination, and condition (4) since the sources of both the added and removed edge are of type $\Sigma^\sharp$. Therefore, $\langle R, C'\rangle$ is well-formed, so by induction $\mathcal{D}'$ is well-formed.

□

Lemma B.2 (DAIG–CFG Consistency Preservation). *If $\mathcal{D} \cong \langle L, E, \ell_0\rangle$ and $\mathcal{D}, M \vdash n \Rightarrow v\,;\mathcal{D}', M'$ then $\mathcal{D}' \cong \langle L, E, \ell_0\rangle$.*

Proof. By cases on the derivation $\mathcal{D}, M \vdash n \Rightarrow v\,;\mathcal{D}', M'$. All cases other than Q-Loop-Unroll are trivial since they don't modify the structure of the CFG.

Q-Loop-Unroll abstractly unrolls a loop body by adding DAIG reference cells and computations with unroll. Since unroll increments the iteration counts of names in the new unrolling, conditions (1) and (2) of 4.2 are vacuously satisfied as they only reference the initial (i.e. zeroth) copy of loop body reference cells. Condition (3) requires that for each fix edge, there exist unrollings of the corresponding loop body's DAIG region for all iterations up to that fix edge's sources. The $i = 0$ through $i = k - 1$ cases are satisfied in the pre-unrolling DAIG, and unroll adds the kth unrolling by incrementing each reference in the $k - 1$th, so Q-Loop-Unroll preserves DAIG–CFG consistency as well. □

Lemma B.3 (DAIG–AI Consistency Preservation). *If $\mathcal{D} \cong \langle \Sigma^\sharp, \varphi_0, [\![\cdot]\!]^\sharp, \sqsubseteq, \sqcup, \nabla\rangle$ and $\mathcal{D}, M \vdash n \Rightarrow v\,;\mathcal{D}', M'$ then $\mathcal{D}' \cong \langle \Sigma^\sharp, \varphi_0, [\![\cdot]\!]^\sharp, \sqsubseteq, \sqcup, \nabla\rangle$*

Proof. By cases on the derivation of $\mathcal{D}, M \vdash n \Rightarrow v\,;\mathcal{D}', M'$. The Q-Reuse and Q-Loop-Unroll cases are trivial because they do not change reference cells contents or add non-empty reference cells. Q-Miss's premises are identical to Definition 4.3's $f \neq$ fix case, and Q-Match's premises, given that $M_k(\underline{f} \cdot (\underline{v_1} \cdots \underline{v_k})) = f(v_1, \ldots, v_k)$ by memoization table soundness, are the same. Q-Loop-Converge's premises are identical to Definition 4.3's $f =$ fix case. □

Theorem B.1 (DAIG From-Scratch Consistency). *For all sound M and well-formed $\mathcal{D}$ such that $\mathcal{D} \cong \langle L, E, \ell_0 \rangle$ and $\mathcal{D} \cong \langle \Sigma^\sharp, \varphi_0, [\![\cdot]\!]^\sharp, \sqsubseteq, \sqcup, \nabla \rangle$, if $\mathcal{D}, M \vdash \underline{\ell} \Rightarrow v \,;\, \mathcal{D}', M'$ then $v = [\![\ell]\!]^{\sharp *}_{\langle L, E, \ell_0 \rangle}$.*

Proof. If the reference cell named by $\underline{\ell}$ is non-empty in $\mathcal{D}$, then Q-Reuse must be the root of the derivation $\mathcal{D}, M \vdash \underline{\ell} \Rightarrow v \,;\, \mathcal{D}', M'$. Therefore, since $\mathcal{D} \cong \langle \Sigma^\sharp, \varphi_0, [\![\cdot]\!]^\sharp, \sqsubseteq, \sqcup, \nabla \rangle$, $v = [\![\ell]\!]^{\sharp *}$.

Otherwise, the reference cell named by $\underline{\ell}$ is empty in $\mathcal{D}$ and we will proceed by cases on ℓ, a non-join location $\ell \in L_{\not\sqcup}$ with CFG in-degree 1, a join location $\ell \in L_\sqcup$ with CFG in-degree ≥ 2, or a loop head ℓ such that there exists a back edge $\ell' \dashv [s] \mapsto \ell \in E_b$. Note that ℓ_0 — the only location that doesn't fall into one of those cases — is excluded because is the reference cell with name $\underline{\ell_0}$ is non-empty by $\mathcal{D} \cong \langle \Sigma^\sharp, \varphi_0, [\![\cdot]\!]^\sharp, \sqsubseteq, \sqcup, \nabla \rangle$.

Case $\ell \in L_{\not\sqcup}$: Since $\mathcal{D} \cong \langle L, E, \ell_0 \rangle$, there exists an $\ell' \dashv [s] \mapsto \ell' \in E_f$, $\underline{\ell'} \cdot \underline{\ell}[s : Stmt] \in R$, and $\underline{\ell} \leftarrow [\![\cdot]\!]^\sharp(\underline{\ell'} \cdot \underline{\ell}, \underline{\ell'}) \in C$. Therefore, the root of the derivation of $\mathcal{D}, M \vdash \underline{\ell} \Rightarrow v \,;\, \mathcal{D}', M'$ is either Q-Miss or Q-Match.

Sub-case Q-Miss: $n_1 = \underline{\ell'} \cdot \underline{\ell}$ and therefore $v_1 = s$ by Q-Reuse. Due to the preceding preservation lemmas and the inductive hypothesis, $\mathcal{D}_1, M_1 \vdash \underline{\ell'} \Rightarrow v_2 \,;\, \mathcal{D}_2, M_2$ and $v_2 = [\![\ell']\!]^{\sharp *}$. Therefore, $v = [\![s]\!]^\sharp [\![\ell']\!]^{\sharp *}$ which, by local abstract interpreter soundness, is equal to $[\![\ell]\!]^{\sharp *}$.

Sub-case Q-Match: By the same arguments as for the Q-Miss case — which follow from premises shared between the two rules — $v_1 = s$ and $v_2 = [\![\ell']\!]^{\sharp *}$. Therefore, by memoization table soundness, $v = M_2(\underline{[\![\cdot]\!]^\sharp} \cdot \underline{s} \cdot \underline{[\![\ell']\!]^{\sharp *}}) = [\![s]\!]^\sharp [\![\ell']\!]^{\sharp *}$ which, by local abstract interpreter soundness, is equal to $[\![\ell]\!]^{\sharp *}$.

Case $\ell \in L_\sqcup$: Since $\mathcal{D} \cong \langle L, E, \ell_0 \rangle$, there are k forward CFG edges into ℓ and for each such $\ell_i \dashv [s_i] \mapsto \ell$, we have $\underline{i} \cdot \underline{\ell_i} \cdot \underline{\ell}[s_i : Stmt] \in R$ and $\underline{i} \cdot \underline{\ell} \leftarrow [\![\cdot]\!]^\sharp(\underline{i} \cdot \underline{\ell_i} \cdot \underline{\ell}, \underline{\ell_i}) \in C$. Also by $\mathcal{D} \cong \langle L, E, \ell_0 \rangle$, there is an edge $\underline{\ell} \leftarrow \sqcup(\underline{1} \cdot \underline{\ell}, \ldots, \underline{k} \cdot \underline{\ell}) \in C$. Therefore, the root of the derivation of $\mathcal{D}, M \vdash \underline{\ell} \Rightarrow v \,;\, \mathcal{D}', M'$ is either Q-Miss or Q-Match. We omit the Q-Match subcase as it is analogous to the Q-Miss subcase in the same manner as the previous ($\ell \in L_{\not\sqcup}$) case.

Sub-case Q-Miss: For each i, $n_i = \underline{i} \cdot \underline{\ell}$. Since $\underline{i} \cdot \underline{\ell_i} \cdot \underline{\ell}[s_i : Stmt] \in R$ and $\underline{i} \cdot \underline{\ell} \leftarrow [\![\cdot]\!]^\sharp(\underline{i} \cdot \underline{\ell_i} \cdot \underline{\ell}, \underline{\ell_i}) \in C$, we know that $v_i = [\![s_i]\!]^\sharp [\![\ell_i]\!]^{\sharp *}$, as the subquery for $\underline{i} \cdot \underline{\ell_i} \cdot \underline{\ell}$ resolves to s_i via Q-Reuse and the subquery for $\underline{\ell_i}$ resolves to $[\![\ell_i]\!]^{\sharp *}$ by the inductive hypothesis. Then, $v = \sqcup([\![s_1]\!]^\sharp [\![\ell_1]\!]^{\sharp *}, \ldots, [\![s_k]\!]^\sharp [\![\ell_k]\!]^{\sharp *})$ which by global abstract interpreter soundness is equal to $[\![\ell]\!]^{\sharp *}$ — the invariant at ℓ is the join of each of its predecessor's invariants, transformed by the abstract semantics of the CFG edge by which they're connected.

Case ℓ is a loop head: Since $\mathcal{D} \cong \langle L, E, \ell_0 \rangle$, ℓ is the destination of a CFG back-edge $\ell' \dashv [s] \mapsto \ell \in E_b$ and $\underline{\ell}$ is the destination of a DAIG fix edge $\underline{\ell} \leftarrow \mathsf{fix}(\underline{\ell}^{(k-1)}, \underline{\ell}^{(k)}) \in C$.

By natural number induction, a query for $\underline{\ell}^{(i)}$ yields the ith abstract iterate at $\underline{\ell}$. In the $i = 0$ base case, a query for $\underline{\ell}^{(0)}$ yields the 0th abstract iterate at ℓ by precisely the argument[8] of the previous two cases, which deal with analysis of forwards dataflow. In the inductive case, a query for $\underline{\ell}^{(k)}$ yields the kth abstract iterate at ℓ, since $\underline{\ell}^{(k)} \leftarrow \nabla(\underline{\ell}^{(k-1)}, \underline{\ell}^{(k-1)} \cdot \underline{\ell}^{(k)}) \in C$ by $\mathcal{D} \cong \langle L, E, \ell_0 \rangle$, and a query for $\underline{\ell}^{(k-1)} \cdot \underline{\ell}^{(k)}$ yields the image of the kth abstract iterate in the abstract semantics of the loop body — as unrolled by definition of unroll — by local abstract interpreter soundness and $\mathcal{D} \cong \langle \Sigma^\sharp, \varphi_0, [\![\cdot]\!]^\sharp, \sqsubseteq, \sqcup, \nabla \rangle$.

The root of the derivation tree of $\mathcal{D}, M \vdash \underline{\ell} \Rightarrow v \,;\, \mathcal{D}', M'$ is either Q-Loop-Converge — in which case the result v of the subqueries for the two most recent abstract iterates are equal and v is therefore the fixed-point $[\![\ell]\!]^{\sharp *}$ — or Q-Loop-Unroll, in which case v is equal to $[\![\ell]\!]^{\sharp *}$ by the inductive hypothesis. □

Theorem B.2 (DAIG Query Termination). *For all M and well formed $\mathcal{D}$ such that $\mathcal{D} \cong \langle L, E, \ell_0 \rangle$ and $\mathcal{D} \cong \langle \Sigma^\sharp, \varphi_0, [\![\cdot]\!]^\sharp, \sqsubseteq, \sqcup, \nabla \rangle$, if n is in the namespace of $\mathcal{D}$ then there exist $v, \mathcal{D}', M'$ such that $\mathcal{D}, M \vdash n \Rightarrow v \,;\, \mathcal{D}', M'$.*

Proof. Let the ancestors of a name n in $\mathcal{D}$ be the set of backwards-reachable names $\{n' \mid n' \rightsquigarrow n\}$. Note that, since $\mathcal{D}$ is well formed and therefore acyclic, if $n \leftarrow f(n_1, \ldots, n_k) \in C$ then each n_i has strictly fewer ancestors than n. We will proceed by induction on the number of ancestors of n.

Base case: Since n has no ancestors, there is no $c \in C$ such that $\mathtt{dest}(c) = n$. Thus, by definition 4.1.5, n names a non-empty reference cell (i.e. $n[v : \tau] \in R$). Therefore, by Q-Reuse, $\mathcal{D}, M \vdash n \Rightarrow v \,;\, \mathcal{D}, M$.

Inductive case: By well-formedness of $\mathcal{D}$ (definition 4.1.2), there is exactly one edge $c = n \leftarrow f(n_1, \ldots, n_k) \in C$ with destination n. We proceed by cases on f.

Case $f \in \{[\![\cdot]\!]^\sharp, \sqcup, \nabla\}$: By repeated application of the inductive hypothesis (along with the preceding preservation lemmas for well-formedness and CFG/AI consistency), for each n_i we may derive $\mathcal{D}_{i-1}, M_{i-1} \vdash n_i \Rightarrow v_i \,;\, \mathcal{D}_i, M_i$, where $\mathcal{D}_0 \triangleq \mathcal{D}$ and $M_0 \triangleq M$.

[8] Modulo the change of name; note that the initial name n_ℓ for a loop head ℓ is $\underline{\ell}^{(0)}$ as opposed to $\underline{\ell}$.

Then, either $\underline{f}\cdot(\underline{v_1}\cdots\underline{v_k}) \in \mathsf{dom}(M_k)$, in which case we apply Q-Match to derive

$$\mathcal{D}_0, M_0 \vdash n \Rightarrow M_k(\underline{f}\cdot(\underline{v_1}\cdots\underline{v_k}))\,;\mathcal{D}_k[n \mapsto M_k(\underline{f}\cdot(\underline{v_1}\cdots\underline{v_k}))], M_k$$

or $\underline{f}\cdot(\underline{v_1}\cdots\underline{v_k}) \notin \mathsf{dom}(M_k)$, in which case we apply Q-Miss to derive

$$\mathcal{D}_0, M_0 \vdash n \Rightarrow v\,;D_k[v/n], M_k; \underline{f}\cdot(\underline{v_1}\cdots\underline{v_k}) \mapsto v$$

where $v = f(v_1, \ldots, v_k)$.

Case f = fix: Since $\mathcal{D} \cong \langle L, E, \ell_0\rangle$, $c = n \leftarrow \mathsf{fix}(\underline{\ell}^{(k-1)}, \underline{\ell}^{(k)})$. By the inductive hypothesis (along with the preceding preservation lemmas for well-formedness and CFG/AI consistency), $\mathcal{D}, M \vdash \underline{\ell}^{(k-1)} \Rightarrow v_1\,;\mathcal{D}', M'$ and $\mathcal{D}', M' \vdash \underline{\ell}^{(k)} \Rightarrow v_2\,;\mathcal{D}'', M''$. If $v_1 = v_2$ then Q-Loop-Converge applies and we derive $\mathcal{D}, M \vdash n \Rightarrow v_1\,;\mathcal{D}'', M''$.

Otherwise, Q-Loop-Unroll applies but, since n has more ancestors in $\mathsf{unroll}(\mathcal{D}'', c)$ than in $\mathcal{D}$, we can not simply apply induction to derive the final premise. Note, however, that the final premise is also a query for n in the unrolled DAIG, which by definition contains the edge $n \leftarrow \mathsf{fix}(\underline{\ell}^{(k)}, \underline{\ell}^{(k+1)})$ and the empty reference cell $n[\varepsilon : \tau]$; therefore, it must be derived either by Q-Loop-Unroll or Q-Loop-Converge. If Q-Loop-Unroll is used then this argument repeats, yielding a tree of nested applications of Q-Loop-Unroll in direct correspondence to the sequence of abstract iterates $\underline{\ell}^{(k)}, \underline{\ell}^{(k+1)}, \ldots$, which is an increasing sequence of abstract states computed by repeated application of ∇. Therefore, by the convergence property of widening in the underlying abstract interpreter, this process may repeat only finitely many times before Q-Loop-Converge applies, yielding a finite derivation tree.

□

References for the Appendix

[4] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition).* Addison-Wesley Longman Publishing Co., Inc., USA, 2006.