Interactive Abstract Interpretation

Thesis Proposal

Benno Stein

January 25, 2021

1 Introduction

Static program analysis is seeing increasingly widespread adoption, with applications both in bug-finding and verification. However, many of the tweaks, implementation hacks, and optimizations applied in practice to enhance usability and scalability are applied in an *ad hoc* manner by analysis designers and accompanied by informal arguments towards correctness, soundness, and termination.

Many analysis designs are based upon the theory of *abstract interpretation* [4], in which an over-approximation of a program's semantics is found by computing a fixed-point over abstractions of the program's concrete values, semantics, and syntax. There are many advantages to this technique, notably:

- **Modularity:** The theory of abstract interpretation is parametric in these abstractions, thus modularizing the design and implementation of program analyses into a series of smaller and more tractable components that can be reused in various combinations. Most commonly, a single fixed-point solver engine is instantiated with many different abstract domains to solve a wide range of analysis problems.
- **Metatheory:** Provided that its components satisfy certain well-understood properties, an abstract interpreter is guaranteed to terminate with a sound result. This has the much-vaunted corollary that analysis results are free of false negatives.

These desirable properties depend on the analysis' global fixed-point, thereby restricting analysis designers to what we call "black-box" analyses: those that hold abstractions constant, take a program as input, and some time later output some facts about that program. However, many modern program analysis problems are incompatible with the restrictions of black-box analysis, so various domain-specific techniques have been developed and applied in practice.

In an IDE, for example, an analyzer must recompute results on the fly as the program is edited by a user, so *incremental* analyses (e.g. [**?**]) that reuse partially computed results and avoid unnecessary re-computation dramatically outperforms a batch analysis that starts anew whenever the program changes.

Analysis results are often only needed at certain locations specified by a client program or human user, in which case a black-box analysis that computes a global fixed-point is wasteful. To that end, *demand-driven* analyses (e.g. [**?**]) respond to extrinsic queries while performing only the minimal amount of analysis computation required.

In some cases, the degree of precision required can vary significantly within a program. Numerous *refinement*-based analyses have been developed to address this problem, either by applying different abstractions to improve precision (e.g. [**? ?**]) or by incorporating information from some oracle (e.g. [**? ? ?**]).

A primary motivation for such non-standard analysis designs is speed: whole-program batch[1] analysis is expensive, and structural changes to analysis architecture are required to deliver results to developers within reasonable timeframes.

This proposal tackles the problem of performing abstract interpretation in real-time, delivering analysis results to developers where and when they are needed in a manner that scales, generalizes to arbitrary abstract domains, and is provably sound and terminating.

1.1 Challenges and Related Work

The key challenge addressed is that efficient incremental and demand-driven analysis require fine-grained tracking of dependencies and caching of intermediate analysis results, but abstract interpretation fixed-point computations induce a complex, cyclic, and unbounded dependency structure.

This unboundedness results from fixed-point computations in infinite-height abstract domains, for which a sequence of widening operations are applied to guarantee termination in some unbounded amount of time, as described by Cousot and Cousot [4] and implemented for example by the chaotic iteration technique of Bourdoncle [3].

As a result, many existing approaches to incremental dataflow analysis (e.g. [1, 6]) and demand-driven dataflow analysis(e.g. [13, 7, 14]) have avoided fully-general abstract interpretation problems in favor of systems such as IFDS/IDE where the domain is assumed to be of finite height. This allows for very efficient incremental or demand-driven analysis, but rules out a large number of important abstract interpretation problems, including the vast majority of numerical (intervals, octagons, polyhedra, etc.) and shape

[1] i.e. neither incremental nor demand-driven

Technique	Infinite domains with widening	Incremental	Demand-Driven
Classic Abstract Interpretation [4, 3]	✓	✗	✗
Incremental Dataflow [1, 6**?**]	✗	✓	✗
Demand-Driven Dataflow [13, 7, 14]	✗	✗	✓
Datalog-Based Analysis [16, 15]	✗	✓	✓
Proposal	✓	✓	✓

Table 1: Feature comparison of existing analysis techniques and this proposal.

domains. Furthermore, all such research of which I am aware has focused on *either* incremental *or* demand-driven analysis.

Some Datalog-based analysis frameworks are capable of both incremental and demand-driven analysis computations [16, 15]. However, like the aforementioned dataflow analyses, the logic programming DSL restricts analysis expressivity and rules out general-purpose abstract interpreters.

1.2 Proposal

Just as a classical batch abstract interpretation engine lifts an abstract domain to a whole-program batch analysis, my aim is to define an abstract interpretation engine capable of computing analysis results in response to arbitrary sequences of program edits and queries quickly enough to support interactive use.

In doing so, I will test the following hypotheses :

Hypothesis 1: The combination of incremental and demand-driven analysis is necessary in order to achieve interactive speeds for generic abstract interpretation problems.

Incrementality and demand are generally thought of and studied as orthogonal: incremental analyses exhaustively analyze a program as edits are issued, while demand-driven analyses respond to queries against a fixed program.

However, the two problems are dual: an incremental analysis discards only those analysis results that depend upon an edit, while a demand-driven analysis computes only those analysis results depended upon by a query. Thus, in the case when an edit has many dependents or a query has many dependencies, incremental-only or demand-driven-only analysis costs approach batch analysis costs.

By combining the two techniques, an analysis engine can mitigate this worst-case behavior and perform the minimal amount of analysis work to soundly handle some given queries and edits.

Hypothesis 2: Formal guarantees of termination, soundness, and from-scratch consistency for interactive analysis can be provided without restricting abstract domain expressivity.

One key advantage of abstract interpretation is its generality and expressivity: because abstract domains need only satisfy some high-level algebraic properties, complex domains have been developed to perform a wide range of analyses.

However, existing techniques (e.g. [**?**]) for incremental and demand-driven analysis place restrictions on the abstract domains – most commonly, that they must be finite or at least of finite height – in order to provide soundness and termination guarantees. This precludes their use for many important analysis problems, including for example interval and shape analysis.

In this thesis, I hope to confirm this hypothesis by developing a formalism and engineering a prototype that combine the desirable metatheoretic properties of classical abstract interpretation with an efficient interactive interface.

Although it may seem esoteric, this has significant practical import since much effort has been devoted to the engineering of abstract domains (e.g. [10**?**]). An interactive engine that shares the standard domain interface and assumptions of classical engines would enable the use of such domains in incremental and/or demand-driven settings.

Hypothesis 3: Graph-based techniques for incremental and demand-driven analysis naturally support extensions for refinement-based analysis.

As I will describe throughout this proposal, the crux of my technique for incremental and demand-driven abstract interpretation is an explicit graph-based representation of partial analysis computations. In this representation, intermediate analysis results and dataflow relationships can be accessed and manipulated effectively as first-class data structures.

The myriad forms of refinement that have been studied in the program analysis literature can thus be represented as extensions to the analysis graph – for example, incorporating dynamic or extrinsically-provided information (as in [**?** **?**]) or adjusting abstractions at analysis time to apply greater precision only where needed (as in [**?**]).

In the remainder of this proposal, I will describe my graph-based technique for interactive abstract interpretation through a motivating example (Section 2), provide details on the underlying theoretical work (Section 3) and prototype implementation (Section 4), and lay out a plan for future work and completion of my dissertation(Section 5).

2 Overview

The core of my approach is a graph-based representation of abstract interpretation computations, generic in both an underlying programming language and an abstract domain. In this section, I will demonstrate how the encoding supports incremental and demand-driven computations by example. For concreteness and clarity, I fix a Java-like imperative language and a separation logic-based shape analysis domain.

Consider the simple imperative program given in Fig. 1, which appends two linked lists. Given null-terminated and acyclic input lists `p` and `q`, `append` must return a similarly well-formed list and not dereference `null` in order to be correct. These properties can be verified using an abstract interpretation-based shape analysis[2, 5, 11], tracking separation logic facts like `lseg`$(p, null)$ to represent well-formedness of list `p`.

Such an analysis, though it may be computed quickly for this small example, can become prohibitively expensive for larger programs and more complex abstract domains. My goal is to compute analysis results in response to queries and program edits with sufficiently low latency to enable real-time *interactive* use, even as programs and domains grow in complexity.

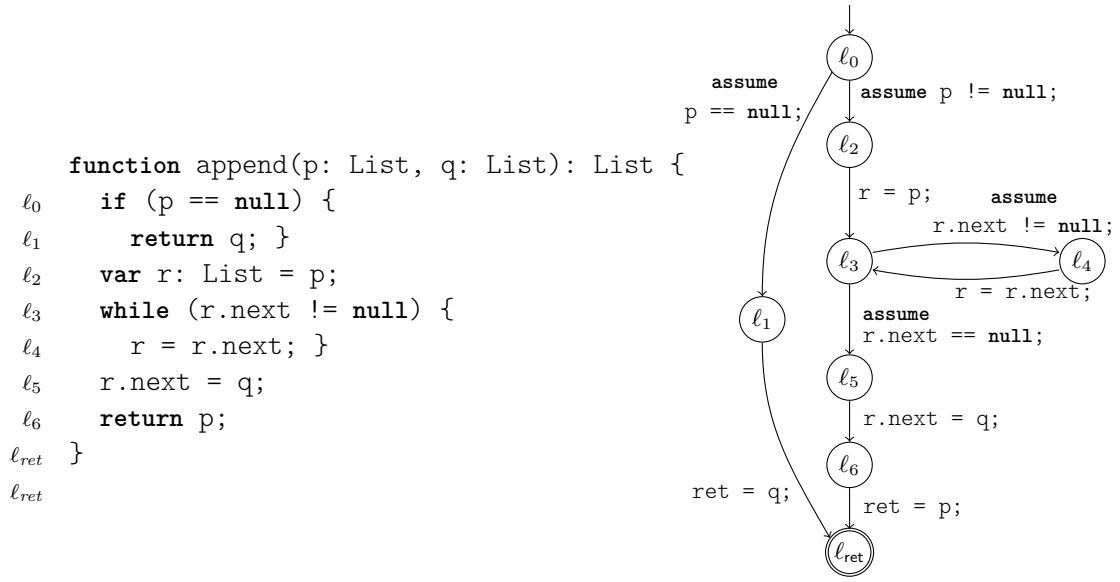


Figure 1: A procedure to append two linked lists in a Java-like imperative language, and an equivalent control-flow graph representation. The labels ℓ_i mark program locations, and loops and conditionals are transformed into unstructured control flow with `assume` statements.

2.1 Reifying Abstract Interpretations

The imperative `append` procedure may equivalently be represented as a control-flow graph (CFG) as shown in Fig. 1, with vertices for program locations and edges labelled by atomic program statements. A classical abstract interpreter analyzes such a program by starting with some initial abstract state 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. 2 reifies the computational structure of such an abstract interpretation over the Fig. 1 CFG. Its vertices are uniquely-named mutable reference cells containing program syntax or intermediate abstract states, and its edges fully specify the computations 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. 2 and throughout this proposal, an underlined symbol denotes the name derived from that symbol: its hash, 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

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

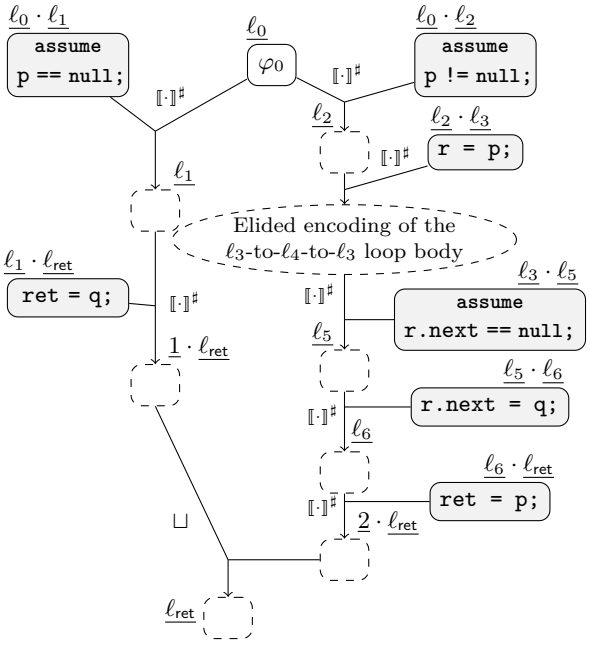


Figure 2: A demanded abstract interpretation graph (DAIG) for the program given in Fig. 1 before any queries are issued. The elided loop encoding is shown in **??**.

over the CFG edge $\ell_0 \to \ell_1$ is encoded in Fig. 2 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. 2) and analysis state (unshaded in Fig. 2) are treated *uniformly* in the DAIG semantics; I differentiate them visually only for clarity. Statement edits and abstract state updates both correspond simply to edits to the relevant mutable reference cells.

2.2 Demand-Driven and Incremental Analysis

In this section, I will show by example how the DAIG encoding naturally supports both demand-driven and incremental analysis. I will use the aforementioned shape-analysis domain: 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 is therefore incompatible with previous frameworks for demand-driven and/or incremental analysis that require finite (or finite-height) domains.

Fig. 5 shows the result (on the affected region only) of evaluating a query on the DAIG given in Fig. 2.

Suppose that a client issues a demand query for the abstract state after the `return q;`

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

statement on line ℓ_1 of the example program; this is the intermediate abstract state named $\underline{1} \cdot \underline{\ell_{\mathsf{ret}}}$ above. Since the $\underline{1} \cdot \underline{\ell_{\mathsf{ret}}}$ cell has predecessors $\underline{\ell_1} \cdot \underline{\ell_{\mathsf{ret}}}$ and $\underline{\ell_1}$, we issue requests for the values of those cells.

Cell $\underline{\ell_1}$ is empty, but depends on $\underline{\ell_0} \cdot \underline{\ell_1}$ and $\underline{\ell_0}$, so more requests are issued. Both of those cells hold values, so we can compute and store the value of $\underline{\ell_1}$. Now, having satisfied its dependencies, we can compute the value of $\underline{1} \cdot \underline{\ell_{\mathsf{ret}}}$.

Note that DAIGs are always acyclic, so this recursive traversal is well-founded.

Crucially, these results are now *memoized* for future reuse; a subsequent query for $\underline{\ell_{\mathsf{ret}}}$, for example, will memo match on $\underline{1} \cdot \underline{\ell_{\mathsf{ret}}}$ and only need to compute $\underline{2} \cdot \underline{\ell_{\mathsf{ret}}}$ 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.

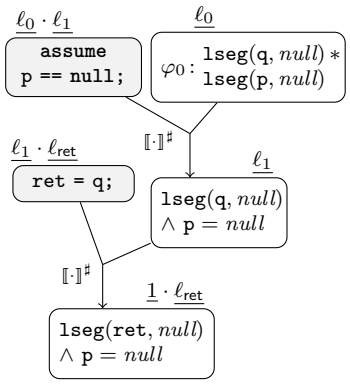


Figure 3: Demand-driven query

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 [8].

Consider for example 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.

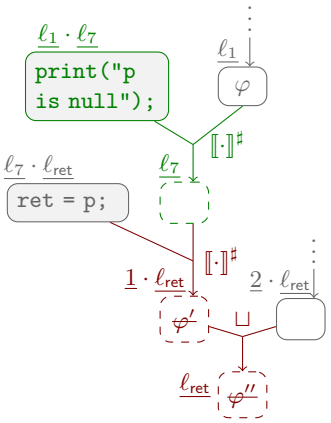


Figure 4: Incremental edit

Fig. 4 illustrates this program edit's effect on the DAIG. The green nodes correspond to the added statement cell $\underline{\ell_1} \cdot \underline{\ell_7}$ and its corresponding abstract state cell $\underline{\ell_7}$.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 $\underline{\ell_1} \cdot \underline{\ell_{\mathsf{ret}}}$ and $\underline{\ell_{\mathsf{ret}}}$ 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. This interplay between eager invalidation and lazy recomputation is key to the efficacy of demanded computation graph-based incremental computation [9], maximizing the sound reuse of intermediate results while minimizing unnecessary computation of unneeded results.

For example, assuming that $\underline{\ell_1}$ and $\underline{2} \cdot \underline{\ell_{\mathsf{ret}}}$ were both computed before the edit, a query for $\underline{\ell_{\mathsf{ret}}}$ must execute only two transfers and one join: the red and green edges of Fig. 4. This represents a significant savings over recomputing the entire analysis — including the loop fixed point — as would be necessary without incremental analysis.

2.3 Dependency Cycles

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 effective demand-driven and incremental abstract interpretation.

For instance, introducing cyclic dependencies in the DAIG yields an unclear evaluation semantics. We can instead enrich the demand-driven query evaluation and the incremental edit semantics to dynamically evolve DAIGs such that each edit/query preserves the DAIG acyclicity invariant. To do so, I 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.

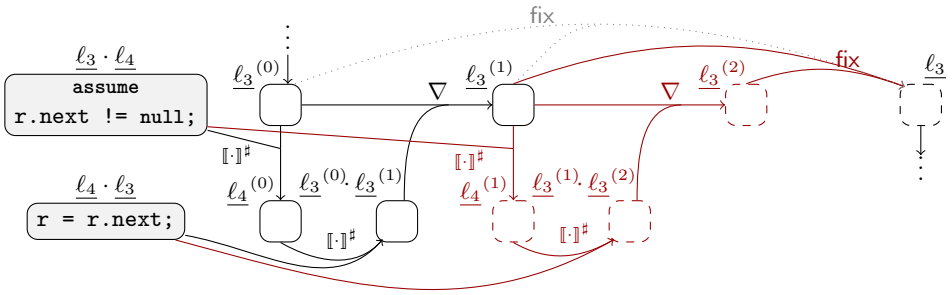


Figure 5: DAIG for the ℓ_3-to-ℓ_4-to-ℓ_3 loop of Section 2 after one unrolling of the abstract interpretation's fixed point computation, with the newly-unrolled region shown in red. Parenthesized superscripts on names uniquely identify successive abstract iterates at each location in the loop, and products of such names are pre-widen intermediate values.

This unrolling procedure operates at the semantic level of the abstract interpretation rather than the syntactic level of the control-flow graph and proceeds one *abstract* iteration at a time until a fixed point is reached, preserving acyclicity at each step. Termination is guaranteed by leveraging the standard argument of abstract interpretation metatheory: the sequence of abstract iterates converges because it is produced by widening a monotonically increasing sequence of abstract states, so the unrolling occurs only finitely — albeit unboundedly — many times.

Since the acyclic DAIG invariant is always preserved, invalidating on incremental edits still only requires eager dirtying forwards in the DAIG, with some special but straightforward semantics for fixed-points designed to maximize reuse.

3 Demanded Abstract Interpretation Graphs

In this section, I summarize the formalization of demanded abstract interpretation graphs (DAIGs) that underpins my approach to interactive abstract interpretation.

3.1 Syntax

As explained informally in the overview, a DAIG reifies the computational structure of an abstract interpretation fixed-point computation as an explicit graph structure. It is parametric in both a programming language and an underlying abstract interpreter:

- Programs under analysis are given as control-flow graphs, edge-labelled by an unspecified statement language and interpreted by a denotational concrete semantics.

statements	$s \in \mathit{Stmt}$	
locations	$\ell \in \mathit{Loc}$	
control-flow edges	$e \in \mathit{Edge}$	$::= \ell \dashv\lfloor s \rfloor\mapsto \ell'$
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	$[\![\cdot]\!]$	$: \mathit{Stmt} \to \Sigma \to \Sigma_\bot$

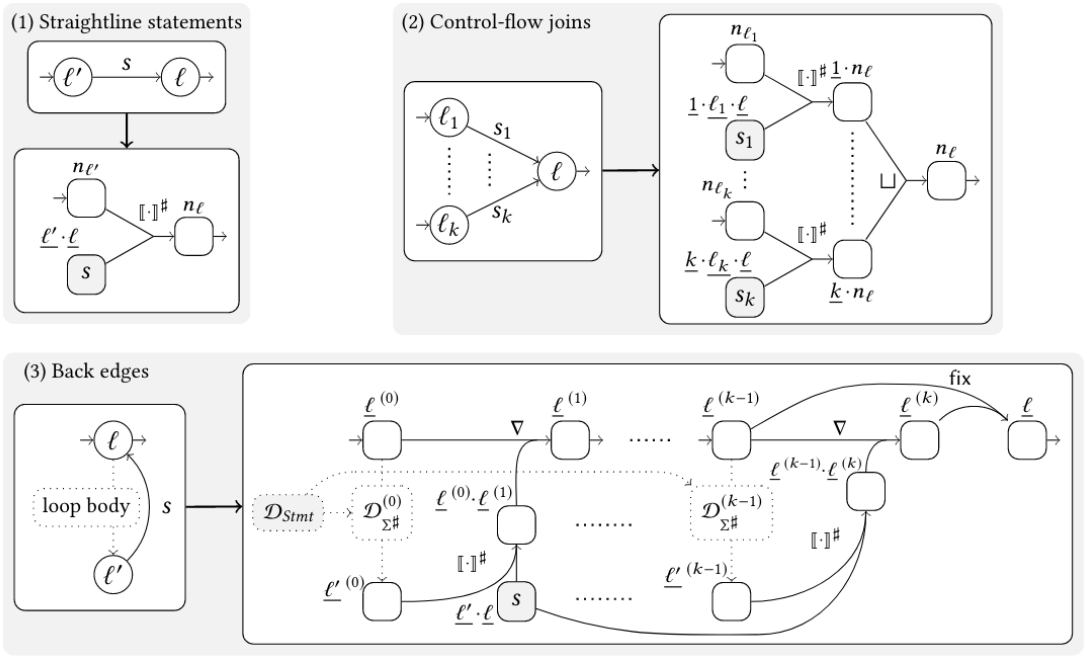
- Abstract interpreters are tuples composed of
  - An *abstract domain* $\Sigma^\sharp$ (elements of which are referred to as *abstract states*) which forms a semi-lattice under
    - a *partial order* $\sqsubseteq \ \in \mathcal{P}(\Sigma^\sharp \times \Sigma^\sharp)$ with a bottom $\bot \in \Sigma^\sharp$
    - 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).

Given these underlying structures, demanded abstract interpretation graphs $\mathcal{D}$ are given by the following grammar:

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

That is, a DAIG $\mathcal{D} = \langle R, C \rangle$ is a directed acyclic hypergraph whose vertices R are *reference cells* (optionally) containing statements and abstract states and whose edges C are *computations* labelled by analysis function symbols, specifying dependencies in the fixed-point computation. Names are also crucially important, acting as memoization keys for analysis inputs and intermediate results.

This definition is carefully designed to encode abstract interpretations of reducible program CFGs, and I have defined and implemented a constructive procedure by which an initial DAIG may be constructed for any such program. The intuition of this procedure is illustrated by the following diagrams[4], which show how each possible CFG structure can be encoded into a DAIG structure:



A forward CFG edge to a non-join location is encoded by a transfer function-labelled DAIG edge connecting reference cells for its abstract pre-state and statement label to a reference cell for its abstract post-state.

Forward CFG edges to join locations are encoded by an analogous 3-reference-cell widget for each incoming edge, with those intermediate results then joined together into the ultimate post-state.

Back edges are encoded by some k (initially 1) unrollings of the abstract interpretation of the corresponding natural loop body, with a widen (∇) edge between each abstract unrolling and a fix edge from the two most recent abstract iterates to the ultimate fixed-point abstract state. Note that this structure is *acyclic* but precisely encodes an unbounded fixed-point computation over a cyclic underlying CFG.

[4] I apply some ad-hoc shorthands in this figure for clarity: n_ℓ is the name of the initial abstract state at location ℓ, $\mathcal{D}_{Stmt}$ contains all of the loop body's statement reference cells, and $\mathcal{D}^{(i)}_{\Sigma^\sharp}$ contains all of its abstract state reference cells with iteration count i. Each dotted line from D_{Stmt} thus represents one or more DAIG edges, from each statement to corresponding abstract states.

3.2 Semantics

A DAIG $\mathcal{D}$ in the above grammar can be seen as a snapshot of a classical abstract interpreter's state, with partially computed results stored in non-empty reference cells and yet-to-be-computed results represented by empty reference cells.

With that view, it is natural to express the evaluation of an abstract interpreter as an operational semantics over DAIGs. I do so with an inductively defined small-step judgment[5] $\mathcal{D} \vdash n \Rightarrow v \;; \mathcal{D}'$, which is read as "requesting the value of cell n in DAIG $\mathcal{D}$ returns v and yields updated DAIG $\mathcal{D}'$." The two key rules defining this judgment are:

$$\text{Q-Reuse}\quad \frac{n[v:\tau] \in R}{\langle R, C\rangle \vdash n \Rightarrow v \;; \langle R, C\rangle}$$

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

The Q-Reuse rule states that when the value of a non-empty cell is requested, that value can be returned and the DAIG left unchanged; Q-Miss handles the case when the value of an empty cell is requested by requesting its dependencies, applying the analysis function f to them, then returning the result and caching it for future reuse.

These evaluation rules govern demand *queries* for the contents of a DAIG, but have no mechanism for handling incremental *edits* to the underlying program or intermediate analysis facts. An edit must update some reference cell and also clear the value of (or "dirty") any reference cell that (transitively) depends on it. This behavior is also defined inductively with a judgment form $\mathcal{D} \vdash n \Leftarrow v_\varepsilon \;; \mathcal{D}'$, read "mutating cell n with value v_ε in a DAIG $\mathcal{D}$ yields updated DAIG $\mathcal{D}'$."

This edit operation is dual to the query operation described above: whereas queries recursively compute the transitive backwards dependencies of some cell, edits recursively dirty the transitive forwards dependencies.

Using these syntactic definitions and operational semantics, I have proven several key properties of DAIGs. The most important top-line results are summarized as follows:

From-Scratch Consistency: For all program locations ℓ, a query for $\underline{\ell}$ in a well-formed[6] DAIG returns $[\![\ell]\!]^{\sharp^*}$, the abstract state at ℓ as computed by a batch abstract interpreter.

Soundness of query results follows directly from this result and the fact the batch analysis is sound. Note that this is in fact a stronger result: not only is the incremental and demand-driven analysis sound, it is exactly as precise as the batch analysis.

Query Termination: For all names n in a well-formed DAIG $\mathcal{D}$, there exists a value v and updated DAIG $\mathcal{D}'$ such that $\mathcal{D} \vdash n \Rightarrow v \;; \mathcal{D}'$.

This is an important result in the context of infinite-height abstract domains, since it is not immediately obvious that the unrolling procedure described above is finite. It relies on the convergence property of the widening operator ∇: since unrolling halts once

[5] This is slightly simplified, eliding the detail of a global memoization table.

[6] Well-formedness here encompasses basic structural properties as well as a relation to some underlying program and abstract interpreter

the sequence of abstract iterates converges and that sequence is produced by applying ∇ to an increasing sequence, the unrolling is guaranteed to terminate.

4 Implementation & Initial Experimental Results

In addition to the formal development described in the previous section, I have built a prototype implementation of an incremental and demand-driven abstract interpretation framework based on DAIGs. I plan to continue extending and improving on this framework as the basis for my future research and dissertation, so will describe it briefly in this section.

The framework is implemented in approximately $2,500$ lines of OCaml code. Incremental and demand-driven analysis logic, including unrolling for fixed-point computations, operates over an explicit graph representation of DAIGs. I also implement global per-function memoization — elided here but formalized in a paper under review — using `adapton.ocaml`, an open-source implementation of the technique of Hammer et al. [8].

The implementation is parametric in an abstract domain, and the effort required to instantiate it with a new domain is comparable to the effort required to do so in a classical abstract interpreter framework. I have implemented several example domains already, including the shape analysis domain used in the overview and two classic numerical domains (Intervals and Octagons [12]), all of which are inexpressible in existing frameworks for incremental and/or demand-driven analysis.

Notably, the numerical domains are backed by APRON [10], a popular library of well optimized off-the-shelf abstract domains. This is an advantage of my approach as compared to existing incremental and demand-driven techniques: since I place no restrictions on the abstract domain, it is compatible existing libraries written in general-purpose programming languages.

This prototype framework has several shortcomings which I plan to address in my thesis research. The most important is its handling of interprocedurality: currently, it supports context-sensitive analysis of non-recursive programs with static calling semantics (i.e., no virtual dispatch or higher-order functions). However, the vast majority of modern programming languages *do* use these features; in order to evaluate the real-world practicality of this technique, it is important that I implement an engine capable of dealing with them. Some details on how I plan to tackle this are in the next section.

4.1 Initial Results

To evaluate the prototype framework, I have performed an experiment comparing its performance against *batch*, *incremental-only*, and *demand-driven-only* analysis. This experiment is conducted in the Octagon abstract domain, over synthetically generated workloads of interleaved analysis queries and edits in a simple imperative JavaScript subset including assignment, arrays, conditional branching, while loops, and non-recursive function calls.

Since the goal of this technique is to provide analysis results to developers interactively and in real-time, I measure the amount of time it takes each analysis variant to produce

analysis results in response to queries.

The following table summarizes my findings across over 150,000 analysis executions:

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

The results indicate that while incremental and demand-driven analysis each significantly improve latencies with respect to the batch analysis baseline, combining the two provides an additional large reduction in analysis 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 program are costly for demand-driven analysis. By combining incremental dirtying with demand-driven evaluation, my technique mitigates these worst-case scenarios and consistently keeps analysis latencies 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.

5 Proposed Future Work

Through this point, this document primarily discusses research that I have already completed. In this section, I will present two extensions to this work that I am currently pursuing and intend to include in my dissertation.

5.1 Higher-Order DAIGs

The statement-generic CFG programming language over which I have described my technique, though it is well-suited for theoretical study and explaining the technique, is insufficiently expressive to encode many real-world programs.

In particular, both *dynamic dispatch* and *recursion* present major issues.

Dynamic dispatch significantly complicates dependency tracking, for both demanded evaluation (since all possible callers of a function must be analyzed to determine the abstract state at that function's entry) and incremental dirtying (since a dirtied callsite's possible callees are unknown). Although recursion can in principle be handled in the same manner as imperative loops[7], it is quite complicated to implement and formally reason about this.

My current and future work attempts to address both of these issues using *higher order* DAIGs. The key idea is to encode the abstract interpretation of an *inter*-procedural pro-

[7] Essentially by unrolling the abstract interpretation of the recursive procedure in the DAIG, naming intermediate values and inserting widens as needed.

gram (with dynamic dispatch and/or recursion) by composing several *intra*-procedural DAIGs using call edges and fixed-point operations.

This technique will enable my prototype analysis tool to scale up to the real-world programs found in public datasets of edits and bugfixes (e.g. [**?**]). Further formal development showing soundness, from-scratch consistency, and query termination of higher-order DAIGs will then form a trustworthy foundation for sound real-time abstract interpretation.

The compositional structure of higher-order DAIGs may also support refinement-based analysis techniques, such as the use of dynamic traces or heuristics to refine call graphs and analysis results (e.g. [**?** **?** **?** **?**]) or the combination of forwards and backwards abstract interpreters for value refinement [**?**].

5.2 Plan for Completion

I plan to perform the research proposed in this document and write my thesis by the end of Summer 2021. I will consider the research complete when I have provided evidence to support each of the hypotheses listed in Section 1.2.

Hypotheses 1 and 2 have already been demonstrated for a restricted class of programs by my paper "Demanded Abstract Interpretation", currently in review at PLDI '21. I plan to submit one to two more publications building upon this work in order to provide evidence that the technique generalizes to more complex "real-world" programs.

First, I will implement the higher-order DAIG structure described in Section 5.1 and build out a more robust frontend that encodes open source programs into these higher-order DAIGs. I will also formalize this analysis by building upon the intraprocedural formalism of the previous paper, showing that query termination and from-scratch consistency are preserved in the lifting. This work is under way and I plan to submit it to either OOPSLA 2021 or POPL 2021.

Then, I plan to investigate hypothesis 3 by extending the higher-order DAIG formalism and analysis framework to incorporate refinement. Ideally, I will be able to instantiate my graph-based abstract interpretation framework to produce variants of existing refinement techniques, including those that rely on dynamic analyses to handle reflection and/or dynamic dispatch [**?** **?**] and those that refine the results of a coarse whole-program analysis with a more precise and targetted analysis [**?** **?**]. I will write a new paper formalizing the integration of refinement into the DAIG framework and submit to SAS 2021 or PLDI 2022.

References

1 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*. ACM, 288–298.

2 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.

3 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.

4 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*. ACM, 238–252.

5 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)*, Vol. 3920. Springer, 287–302.

6 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*. ACM, 307–317.

7 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)*.

8 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.

9 Matthew A. Hammer, Yit Phang Khoo, Michael Hicks, and Jeffrey S. Foster. 2014. Adapton: Composable, Demand-driven Incremental Computation. In *PLDI*. ACM, 156–166.

10 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.

11 Stephen Magill, Aleksandar Nanevski, Edmund Clarke, and Peter Lee. 2006. Inferring Invariants in Separation Logic for Imperative List-processing Programs. In *SPACE*. Citeseer.

12 Antoine Miné. 2006. The Octagon Abstract Domain. *High. Order Symb. Comput.* 19, 1 (2006), 31–100.

13 Thomas W. Reps. 1994. Solving Demand Versions of Interprocedural Analysis Problems. In *CC*.

14 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.

15 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.

16 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*. ACM, 320–331.