Demanded Abstract Interpretation

by

Benno Stein

B.A., Williams College, 2015

M.S., University of Colorado Boulder, 2017

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

2023

Committee Members:

Bor-Yuh Evan Chang, Chair

Gowtham Kaki

Sriram Sankaranarayanan

Fabio Somenzi

Manu Sridharan

David Van Horn

Stein, Benno (Ph.D., Computer Science)

Demanded Abstract Interpretation

Thesis directed by Prof. Bor-Yuh Evan Chang

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 for a developer to get analysis results after a code change. This dissertation considers the problem of making expressive and sophisticated static analyzers *interactive* by providing analysis results to developers in as close to real time as possible. While existing techniques offer some demand-driven *or* incremental aspects for certain classes of analysis, the fundamental challenge addressed by this work is doing *both* for abstract interpretation in arbitrary domains.

This dissertation presents a technique, *demanded abstract interpretation*, that lifts analysis computations to a dependency graph structure in which incremental program edits and demand-driven evaluation of abstract semantics can be handled uniformly. Demanded abstract interpretation draws inspiration from graph-based approaches to incremental computation, and is not only sound and terminating but also *from-scratch consistent* with underlying batch analyses.

The approach is parametric in the choice of abstract domain, supporting a wide range of analysis problems and enabling the reuse of highly-tuned existing domain implementations in our demanded analysis framework without requiring any per-domain reasoning about incrementality or demand. The complex, cyclic, and unbounded dependency structures that arise when analyzing loops and recursive control flow in an infinite-height domain are a key challenge, which our approach handles by dynamically extending novel acyclic encodings of such analysis computation.

This dissertation describes and formalizes demanded abstract interpretation techniques for both intraprocedural analysis and compositional interprocedural analysis. We also present promising experimental results in a prototype analysis implementation, and describe some extensions to the framework designed to confront practical resource constraints without sacrificing formal guarantees.

Acknowledgements

I am immensely grateful to the mentors, colleagues, friends and family who helped me prepare for and write this dissertation. Without their support, I never would have made it through the highs and lows that have made this journey so rewarding.

First and foremost, I am deeply indebted to my advisor Evan Chang, whose sharp technical insights and clarity of thought have shaped me as a researcher and whose mentorship has been indispensable. Evan's commitment to his students' growth and success is unmatched, and his confidence in me has meant more than he knows. I am also grateful to Manu Sridharan for his support and guidance, his always-discerning advice and comments, and his many contributions to the work in this dissertation. Thank you to Matthew Hammer, whose perspective and friendship helped me out of the most difficult time in my graduate studies and set me on the path that would become my thesis topic.

Thank you to the collaborators and colleagues who broadened my perspectives and helped show me the ropes of research and engineering: Lazaro Clapp, Benjamin Barslev Nielsen, Esben Andreasen, Anders Möller, and Kevin Bierhoff. I also benefitted greatly from the insight and support of Peter O'Hearn, Jules Villard, Scott Owens, and especially Josh Berdine, all of whose perspectives have informed my thinking and my approach to research.

The CUPLV group has also been a great source of community, camaraderie, and learning for me over the course of my time in Colorado. Thank you to labmates and friends past and present for supporting me with great feedback on works-in-progress and sharing many a meal, drink, and good conversation, particularly Tianhan, Shawn, Nick, Souradeep, Taisa, Saeid, Aleks, Sergio, Jared,

Jack, Mateo, Vishnu, and Prasanth.

Thank you from the bottom of my heart to my family for helping me get here and supporting me throughout. To my parents, thank you for your endless belief in me and the encouragement whenever I needed it. To Nat and Elena, I've always known I could rely on your perspective and your advice and it's meant a lot to me to share so much of this chapter with you. Finally, to Lucie, your love and support have kept me afloat through the most difficult challenges I've faced. You were always there for me no matter where I was or how I was doing, and I could not have done this without you.

Contents

Chapter

Tables

Table

Figures

Figure

Chapter 1

Introduction

Static program analysis tools are seeing increasingly widespread adoption for verification and bug-finding, but their ergonomics and user experience leave something to be desired. Static analyzers — typically designed to validate a program before it is deployed or merged — can take minutes or hours to report findings to developers. As such, a developer may have already moved on to some other task before analysis completes, causing them to miss or ignore analysis results or to undergo a frustrating and slow process of context-switching between activities and waiting for results. Context-switching of this form is widely recognized as a major inefficiency in the software development process (Tregubov et al., 2017; Sedano et al., 2017).

Nonetheless, the benefits of program analysis are worth the wait: sound static analyses can provide valuable assurances of both functional correctness and safety, even for large and complicated software systems (Sadowski et al., 2018; Distefano et al., 2019).

Much existing work on program analysis is based on *abstract interpretation* (Cousot and Cousot, 1977), an expressive and general framework for designing, building, and reasoning about program analysis tools. At a high level, an abstract interpreter computes an over-approximation of a program's state space by evaluating the program in some *abstract domain*, elements of which (known as abstract states) potentially represent many concrete program states. Abstract domains are equipped with operations to abstract program semantics (i.e. a "transfer function") and merge abstract states (i.e. "join" and/or "widen"). An abstract interpreter can soundly and efficiently prove properties that hold for *all* possible concrete executions of a program by computing a fixed-point

over the program's control-flow graph in a suitable abstract domain.

There are many advantages to this technique, notably its modularity and its robust metatheory:

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 different 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: if an abstract interpreter reports that a program is "safe" with respect to some invariant, then that invariant will never be violated at run time.

These desirable properties depend on the abstract interpreter reaching a global fixed-point, thereby restricting analysis designers to *batch* analyses which take a program as input, compute such a fixed-point, and some time later output facts about that program. Unfortunately, as programs grow in size and domains in complexity, batch abstract interpretation can become extremely costly.

A vast body of research on abstract interpretation has developed myriad abstract domains, combinators thereof and analysis engines, and applied them to reason about program safety and correctness properties. Although these developments have made it possible to analyze increasingly large programs, even the most advanced batch abstract interpreter deployments takes on the order of 15 minutes to hours to do so (Distefano et al., 2019).

As such, there has also been a great deal of research into non-batch analysis infrastructure. 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. (Arzt and Bodden, 2014; Do et al., 2017; Ryder, 1983)) that reuse partially computed results and avoid unnecessary re-computation can dramatically outperform 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 batch analysis that computes a global fixed-point is wasteful. To that end, *demand-driven* analyses (e.g. (Reps, 1994; Duesterwald et al., 1995; Sagiv et al., 1996)) respond to extrinsic queries for specific analysis facts while performing only the minimal amount of analysis computation required.

This dissertation explores a unified approach to improving the responsiveness and interactivity of program analysis tools, built on the foundations of *incremental computation*. Rather than improving the efficiency of traditional batch abstract interpretation, we present and argue for a more expressive interface with first-class notions of program *edits* and client-issued *queries* for analysis results. Leveraging the theory of abstract interpretation and applying insights from the general incremental computation literature, we describe and implement a framework for real-time-interactive program analysis that maintains the robust formal guarantees and domain-agnosticity of abstract interpretation.

Unlike prior approaches to incremental and/or demand-driven program analysis, this framework (a) is fully general with respect to abstract domain, neither placing restrictions on analysis designers nor requiring them to reason about incrementality/demand; and (b) yields results that are provably exactly as precise as those produced by the underlying batch analysis.

1.1 Thesis Statement

The goal of this dissertation is to outfit arbitrary abstract interpreters with a more expressive and user-aware interface tailored to real-world software development workflows, without compromising domain modularity or weakening guarantees of soundness, termination or precision.

> Graph-based representations of analysis state with fine-grained and dynamically-updated dependencies provide a basis for real-time user interaction with abstract interpreters in arbitrary domains.

Our approach achieves fine-grained incrementality and demand by *reifying* abstract interpreta-

tion computations as dependency graphs, in which program syntax and intermediate analysis results are cached for future reuse and explicitly connected by edges denoting analysis data dependencies.

Drawing insight from general incremental computation techniques for interactive systems (Hammer et al., 2014, 2015), this explicit graph representation of analysis computations naturally enables both incremental edits (by "dirtying" analysis results reachable from the edit) and demand-driven query evaluation (by computing backwards-reachable dependencies as needed). Notably, this approach is provably *from-scratch consistent* with the underlying batch analysis, meaning that it produces the exact same analysis results that a batch analyzer would, were it to be run "from scratch".

A key challenge in applying these graph-based techniques in program analysis is handling the complex, cyclic, and unbounded dependency structure induced by abstract interpretation of programs with loops and/or recursion. As we describe in Section 5.3.2 and Section 6.2 for loops and recursion respectively, our approach preserves the acyclicity and finiteness of its dependency graphs by dynamically updating dependencies at analysis time.

1.2 Summary of Contributions

This dissertation is organized as follows: Chapter 2 motivates the need for more interactive analysis tools by case study, describing a typical user experience with state-of-the-art static analysis and our "vision" for a better alternative. In Chapter 3, we describe the philosophy of this dissertation's approach and the foundations on which it builds, then place it in the context of related work and existing techniques.

Chapter 4 provides an overview of the key ideas of this dissertation. It also presents some challenging example programs and analysis problems to illustrate the shortcomings of existing techniques and works through those examples to build intuition about how our technique works.

In Chapter 5 we present *demanded abstract interpretation graphs* (DAIGs), describe how they can be used to automatically provide incremental and demand-driven interfaces for arbitrary *intra*-procedural abstract interpreters, and provide experimental evidence of their expressivity and

scalability.

Building on the intraprocedural building blocks of the DAIG framework, Chapter 6 presents *demanded summarization graphs* (DSGs), which generalize the technique to summary-based *inter*-procedural abstract interpretation. In particular, we show how tabulation-style analysis infrastructures are particularly amenable to incremental and demand-driven evaluation, without limiting abstract domain expressivity.

Chapter 7 walks through some extensions and variants of DSGs, addressing the most pressing practical considerations for implementations and walking through their impacts (or lack thereof) on the formal guarantees of the framework. Finally, Chapter 8 briefly presents some conclusions and possible avenues for future work.

Chapter 2

Motivation: Interactive Program Analysis

This chapter motivates the development of interactive analysis frameworks and formalisms through a simplified example, in which a developer at a large software organization with state-of-the-art analysis infrastructure attempts to implement and verify a new feature. We consider the *user experience* of a non-expert user of program analysis tooling here rather than any specific internal analysis details.

2.1 State of the Art: Analysis in Continuous Integration

Consider a typical developer working in a large software organization, tasked with adding some functionality or feature to a narrow corner of an important piece of software (say, the `WidgetFactory` component of App).

As a first step, they check out the latest development version of the application, which we'll call App_0. After diligently and carefully implementing the new `WidgetFactory` feature, they commit the changes and upload them to central code review/continuous integration (CI) infrastructure, yielding a new version App_1 of App. This is illustrated in the first four lines of the "Developer" column of Fig. 2.1, up to the first `git push`: the developer creates a new feature branch, implements their feature, then commits and pushes the changes.

Once the CI server receives the proposed new version App_1, it runs it through a sophisticated regimen of static analyses to ensure that various safety and correctness invariants are maintained. Depending on the size of the app and the nature of the analyses being performed, this can take

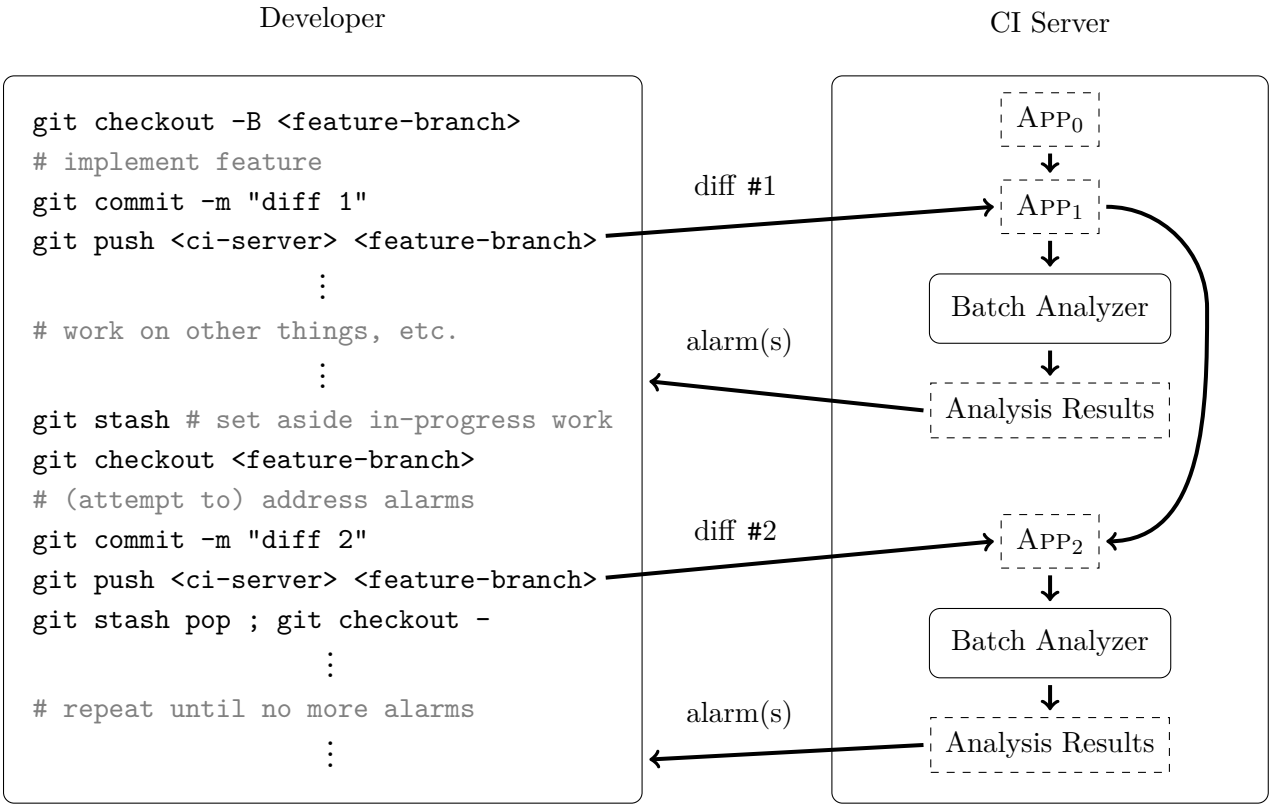


Figure 2.1: Static analysis in a typical continuous integration (CI) deployment: a developer sends some changes (a "diff") to the CI server, which analyzes the application and (eventually) responds with some alarm(s). If/when more changes are made and sent to CI, the entire analysis is re-run from scratch on the new program version.

anywhere from a few minutes to several hours.

Because it takes so long, our developer likely switches to some other activity while waiting for results. Suppose that 90 minutes after pushing diff `#1`, they receive an "alarm": a static analysis tool has identified a potential bug introduced elsewhere in App by the changes to `WidgetFactory`.

Our hypothetical developer now must "context switch" back from whatever they were doing to the `WidgetFactory`, address the alarm(s), and push an updated version App_2 to CI, as depicted in Fig. 2.1.

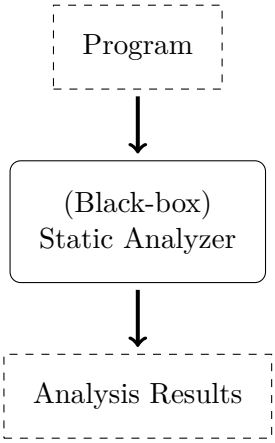
The automated CI system again runs a battery of tests and analyses on the app, then this cycle repeats until no alarms are issued and the new features can safely land on the main development branch.

In a sense, this is the system working as intended: the analysis tool ensured that potentially-buggy code didn't land and affect other developers or users of App. However, that assurance came at the (human) cost of several frustrating and slow cycles of chasing down analysis alarms and waiting for results and the (computational) cost of reanalyzing the entire application multiple times, even though the code changes were relatively small and the area of concern was known after the first alarm was raised on App_1.

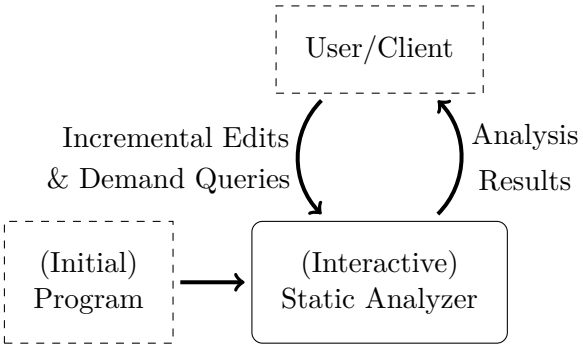
2.2 Vision: Interactive Static Analysis

The core goal of this dissertation is to minimize these two inefficiencies with next-generation analysis infrastructure designed to exploit redundancies while re-analyzing different versions of a mostly-similar program and to give the developer more control over how and where they want analysis to run.

Were this vision to be fully realized, the developer experience described above would proceed quite differently. Rather than analyzing App_1 from scratch and in its entirety in CI, an *incremental* static analyzer can use previously-computed analysis results from App_0, reanalyzing only those portions of the app that are affected by diff `#1`. Since a single diff is quite small relative to the size of the full application, this reanalysis is also proportionally inexpensive. As such, App_1 can



(a) "Black-box" program analysis paradigm, in which a program analyzer is an opaque function from whole-program inputs to analysis fact outputs.



(b) "Interactive" program analysis paradigm, supporting arbitrarily interleaved incremental edits and demand queries against a static program analyzer.

Figure 2.2: Black-box vs. interactive program analysis interfaces: traditional implementations and formalisms are black-box, while this dissertation proposes interactive variants.

effectively be analyzed locally before the diff is even committed or pushed, producing the alarm in real-time. After the developer makes changes to address the alarm, they can now re-analyze the location where it was issued on *demand*, thus verifying that their changes indeed fix the potential bug flagged by the initial alarm.

The key obstacle to real-time user interaction with static analysis tools is the *black-box* nature of their implementations and formalisms, as illustrated in Fig. 2.2a. Modern research on static analysis has successfully scaled such analyzers up to handle massive programs (Blanchet et al., 2003; Cousot et al., 2009; Distefano et al., 2019). Although this has enabled the CI workflow described in Section 2.1, real-time-interactive speed remains far out of reach.

Instead of attempting to further improve performance and scalability of black-box analysis, we propose a new analysis infrastructure paradigm: one with more fine-grained modes of user interaction tailored to real-world developer workflows, as depicted in Fig. 2.2b. In particular, this dissertation leverages two complementary insights about software development, namely:

- *Incremental Edits*: Software development and maintenance naturally consists of many iterative changes to an existing system, with major restructurings few and far between. As such, the successive runs of a batch analysis depicted in Fig. 2.1 constitute an immense amount of redundant and repetitive analysis of unchanged code.

- *Demand-driven Queries*: When a particular developer's changes are limited in scope or they have already verified (formally or informally) some regions of the program, they may only be interested in some particular subset of analysis results. In this case, it is overkill to compute a global over-approximation of program behavior to verify some safety property of the entire system.

Chapter 3

Philosophy of Approach & Related Work

In this chapter we elaborate on the claims and terminology of the introductory chapter by giving a more technical account of this dissertation's motivation and positioning in the context of related work.

First, we describe the philosophy of this dissertation's approach in Section 3.1: why its particular combination of techniques and features is chosen and how they enable the vision for interactive program analysis tools laid out in Chapter 2. Then, we make a detailed tour of related work, covering the foundational pillars of our approach in Section 3.2.1 and Section 3.2.2 before comparing and contrasting it with related techniques in Section 3.3.

3.1 Philosophy of Approach

This section gives a high-level account of the technical challenges addressed by this dissertation and how its contributions enable interactive program analysis infrastructure.

Our primary goal is to perform abstract interpretation in real-time, delivering analysis results to developers where and when they are needed in a manner that (1) generalizes to arbitrary to abstract domains, (2) preserves the robust metatheoretic guarantees of the underlying abstract interpretation framework, and (3) scales up to large real-world programs.

(1) *Generality:* The abstract interpretation framwework (Cousot and Cousot, 1977) computes an over-approximation of a program's concrete semantics by computing a fixed-point over *abstractions* of the program's concrete values, semantics, and syntax. One key advantage of

this approach is its modularity: since the theory is parametric in these abstractions, the design and implementation of program analyses is reduced into a series of smaller and more tractable constituent tasks (Cousot and Cousot, 1979). Most commonly, this is manifested in the form of a single fixed-point solver engine that can be instantiated with many different abstract domains to solve a wide range of analysis problems. Similarly, common abstract domains can be packaged and distributed as libraries for use in various solvers and analyzer infrastructures (Jeannet and Miné, 2009; Singh et al., 2017).

However, these plug-and-play analysis implementations are invariably black-box *batch* engines in the sense of Fig. 2.2a, as they rely on the classic Knaster-Tarski fixpoint theorem (Tarski, 1955) to ensure soundness and termination. This dissertation aims to provide an *interactive* analysis engine (as in Fig. 2.2b) that is *as general* with respect to abstract domain as the standard batch engines.

In practical terms, this means that our formalism and implementations are parametric in abstract domains that are as standard as possible, requiring no domain-specific machinery for incrementality or demand. As we will discuss in Section 3.3, this is in contrast with existing incremental and demand-driven analyzers which restrict abstract domain expressivity and/or employ ad-hoc mechanisms for interactivity.

(2) *Metatheory:* Moreover, this interactive analysis engine that we set out to design and implement ought to provide the same — or as near as possible — metatheoretic guarantees that have made abstract interpretation such a widely used and powerful framework for designing and building program analyses.

There are two main properties of interest for analysis designers: soundness and termination. A *sound* analysis is one that computes an over-approximation of all possible concrete program behaviors. This is an essential property for any analysis that is used to perform compiler optimizations or verify critical safety properties, as it carries the much-vaunted corollary that the analysis has “no false negatives”. Thus, program optimizations or transformations can be

performed without risk and entire classes of bugs can be ruled out before the program is ever run.

A *terminating* analysis does what it says on the tin: terminates in finite time on any finite input program. For abstract domains that form lattices of finite height, termination follows directly from the Knaster-Tarski fixpoint theorem (Tarski, 1955); essentially, since programs are finite and transfer functions are monotone, the analysis can only take finitely many steps before reaching the $\top$ ("top") element of the abstract domain at every program location.

However, some additional machinery is required for abstract domains of infinite height. Namely, a widening operator ∇ is applied on all control-flow cycles; when this operator satisfies the "ascending chain condition"[1] , analysis convergence and therefore termination is guaranteed (Cousot and Cousot, 1977).

There are well-known batch algorithms to compute the solution of an abstract interpretation problem in a manner that is both sound and terminating. The textbook example, "chaotic iteration", keeps a work-list, popping arbitrarily-chosen elements thereof and applying the requisite transfer function, join, and/or widen until a fixed point is reached (Cousot, 1977). In practice, there are more efficient strategies to compute a fixed point, for example by computing a weak topological ordering of program locations and applying semantic functions according to that order (Bourdoncle, 1993).

To our knowledge there is no *general* approach to either incremental or demand-driven abstract interpretation that is provably sound and terminating.[2]

(3) *From-scratch Consistency:* In addition to satisfying the core abstract interpretation metatheoretic properties, it is important that our interactive analysis framework behaves consistently and correctly in the presence of incremental program edits and demand queries.

To that end, we carefully design a system that is "from-scratch consistent" with the underlying

[1] i.e. for all ascending sequences $\varphi_0 \sqsubseteq \varphi_1 \sqsubseteq \varphi_2 \sqsubseteq \ldots$ in the abstract domain, the sequence $\varphi_0,\ \varphi_0 \nabla \varphi_1,\ (\varphi_0 \nabla \varphi_1) \nabla \varphi_2, \ldots$ converges

[2] With the exception of the author's own work (Stein et al., 2021a), which will be discussed in Chapter 5

batch abstract interpreter, i.e. such that incrementally-computed query results are identical to those that would be computed by the batch analyzer were it to be run from scratch on the current program version. Although from-scratch consistency is the "fundamental correctness property of incremental computation" (Hammer et al., 2015), no existing incremental or demand-driven analysis has been proven from-scratch consistent to our knowledge [2] and many in fact violate the condition in order to make use of sound but potentially less precise analysis results computed for previous program versions (Szabó et al., 2016, 2018).

In the context of program analysis, from-scratch consistency ensures not only that analysis results are sound (since the underlying batch analysis is sound) but also that there is no loss of precision due to the incremental or demand-driven solver engine. As a result, our analysis framework can operate as a drop-in replacement for a batch abstract interpretation engine without requiring any ad-hoc reasoning about incrementality or demand by analysis designers or users.

3.2 Related Work

Having now described the goals (and non-goals) of this dissertation, this section gives an overview of the foundational work upon which we build, focusing separately on abstract interpretation (Section 3.2.1) and incremental computation (Section 3.2.2). Then, we lay out a taxonomy of existing work in the broad area of interactive and user-aware program analysis, describing some pros and cons of prior art and positioning this dissertation in necessary context.

3.2.1 Abstract Interpretation

Abstract interpretation is an extremely general and flexible theoretical framework for reasoning about the semantics of programs (Cousot and Cousot, 1977, 1979). Cousot and Cousot (1977) open their seminal paper with the following succinct description:

> A program denotes computations in some universe of objects. Abstract interpretation of programs consists in using that denotation to describe computations in another

> universe of abstract objects, so that the results of abstract execution give some informations on the actual computation.

At the core of abstract interpretation is abstraction: the relation between some concrete interpreter of programs and an abstract counterpart. In this dissertation, we consider programs as *control-flow graphs* (CFGs): directed graphs whose vertices are program locations ℓ and whose edges are labelled by program statements s.

A *concrete interpreter* is a state transformer: at each step, some CFG edge is traversed, and its label s is interpreted in the current state σ to produce the next state σ'. These concrete states each represent a single memory and environment configuration of the program, and the "denotation" $[\![s]\!]$ of a statement s is a function over concrete states, i.e. $[\![s]\!]\sigma = \sigma'$.

An *abstract interpreter* is an analogous state transformer: at each step, some CFG edge is traversed, and its label s is interpreted in the current *abstract* state φ to produce the next *abstract* state φ'. These abstract states are mathematical objects that each represent some set of concrete states, allowing the abstract interpreter to simultaneously reason about the potentially-unbounded space of concrete program executions, using an alternative denotation $[\![s]\!]^\sharp$ of statements as functions over abstract states, i.e. $[\![s]\!]^\sharp\varphi = \varphi'$.

So, given "computations in some universe of objects" (concrete interpreters) and "computations in another universe of abstract objects" (abstract interpreters), what does it mean for "the results of abstract execution [to] give some informations on the actual computation"?

The desired property is known as *soundness* and essentially requires that the reachable states of the abstract interpreter represent all reachable states of the concrete interpreter[3] . A key result of Cousot and Cousot (1977) is that this "global" soundness property is implied by a much simpler "local" soundness property. An abstract interpreter is locally sound if, whenever it takes a step from φ to $\varphi' = [\![s]\!]\varphi$ over some statement s, the concrete interpretation $\sigma' = [\![s]\!]\sigma$ of every concrete state σ modelled by φ is modelled by φ'. Thus, the designer of an abstract domain needs only ensure that

[3] We say that a concrete state σ is represented or "modelled" by an abstract state φ, written $\varphi \models \sigma$ or equivalently that φ "abstracts" σ, when that concrete state is included in the set of concete states represented by the abstract state, written $\sigma \in \gamma(\varphi)$

their abstract transfer function $[\![\cdot]\!]^\sharp$ is locally sound in order to construct an analysis that soundly over-approximates all possible concrete program behaviors.

This mathematical foundation supports a wide range of different abstractions, designed to reason about many different properties of programs, including for example:

- Compiler dataflow analyses (such as reaching definitions, live variables, available expressions, and constant propagation) whose abstract states track some set of statically-known low-level facts about program execution (e.g. which variables have constant value, which variables may be used later in the program, which arithmetic expressions have been previously computed) (Aho et al., 2006; Kam and Ullman, 1977, 1976; Kildall, 1973).

- Numerical analyses, whose abstract states represent some constraints on the values of the numerical variables of the program. Classic examples include sign, parity, and interval analysis, which abstract numerical values respectively by their positive/negative/zero sign, their odd/even parity, and upper/lower bounds (Cousot and Cousot, 1979, 1977). Richer numerical domains such as octagons, polyhedra, and zonotopes allow for more precise abstraction of numerical constraints relating multiple program variables, and can be used to bound the reachable state space of complex systems, prove loop invariants/termination, or statically discharge array-bounds checks (Cousot and Halbwachs, 1978; Jeannet and Miné, 2009; Singh et al., 2017; Goubault and Putot, 2015; Goubault et al., 2012).

- Shape analyses, designed to reason about pointers and dynamically-allocated heap structures such as linked lists and trees. Many abstract domains for shape analysis are based on separation logic or three-valued logic, which allow for the abstraction of many concrete heap configurations in a succinct manner and can be used to verify memory-safety invariants (Berdine et al., 2005; Rinetzky and Sagiv, 2001; Distefano et al., 2006; Calcagno et al., 2011; Chang et al., 2007; Chang and Rival, 2008).

- Domain combinators/products, which allow for principled reasoning across multiple cooperating domains for improved efficiency and/or precision. The most straightforward such combinator is

the direct (a.k.a. cartesian) product (Cousot and Cousot, 1979), which combines abstract states from multiple domains and applies domain operations pointwise. The reduced product (Cousot and Cousot, 1979) is similar, but refines the domain operations by combining information from the constituent abstract domains, while the logical product (Gulwani and Tiwari, 2006; Cousot et al., 2011) shares equalities and logical constraints across its components in a manner similar to the Nelson-Oppen method for combining decision procedures (Nelson and Oppen, 1979).

These examples are intended to demonstrate the generality and expressivity of the abstract interpretation framework, and are far from exhaustive — there is a massive body of research on different abstract domains designed to reason about myriad properties of programs.

This expressivity and generality motivates this dissertation: given the breadth of tools that have been developed in the abstract domain literature, we aim to use bog-standard abstract domains and preserve the metatheoretic guarantees of typical batch engines while providing an interface more tailored to user interaction and the software development workflow.

3.2.2 Incremental Computation

Incremental computation is the problem of efficiently handling changes to a program's input. Consider for example a batch computation f which maps inputs to outputs, as depicted in the solid lines of Fig. 3.1.

When the input changes from in_0 to in_1, the new output can of course be computed by applying f to the new input. However, this *batch* recomputation ($\text{out}_1 = f(\text{in}_1)$) likely has significant overlap with the initial computation ($\text{out}_0 = f(\text{in}_0)$) under the assumption that a relatively small change in input Δ_{in} produces a correspondingly small change in output Δ_{out}.

As such, it is natural to instead compute the new output *incrementally*, reusing as much of out_0 as possible to find out_1 rather than throwing it away and computing $f(\text{in}_1)$ from scratch.

Techniques to do so in a general manner (i.e. for arbitrary f) have been studied in many different forms over the years.

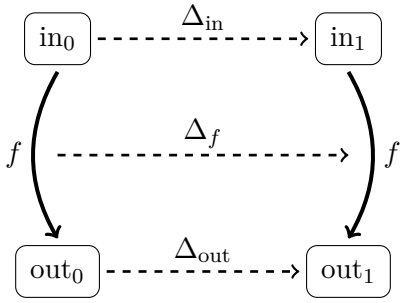


Figure 3.1: Generic incremental computation: given a batch computation f from inputs to outputs, how can we efficiently update outputs (i.e. compute Δ_{out}) in response to input changes Δ_{in}?

Computations using pure functions are particularly amenable to incrementalization using various forms of caching and memoization (Abadi et al., 1996; Field and Teitelbaum, 1990; Pugh and Teitelbaum, 1989). Techniques based on explicit dependency graphs have also been developed, with applications to analysis problems in syntax-directed editors (Demers et al., 1981; Reps, 1982).

More recent work on "adaptive" or "self-adjusting computation" has built on and improved these generic memoization and dependency graph-based approaches. Such techniques have been applied in both functional (Acar et al., 2002; Chen et al., 2014b,a) and imperative (Acar et al., 2008; Hammer et al., 2011, 2009) settings, providing language primitives to support fine-grained automatic caching and reuse in the presence of changes to program inputs and/or an underlying data store.

Although these techniques are quite effective at improving the incremental performance of programs with changing inputs, they are eager in that they update outputs automatically in response to input changes. This can lead to needless recomputation when some or all outputs are no longer needed and hinders the composability, reusability, and interoperability of incremental computations. For interactive systems, lazy on-demand recomputation can avoid both of these pitfalls.

Recent work has explored this approach using language primitives for mutable reference cells, suspended computations (i.e. *thunks*), and a lazy demand-driven semantics for "forcing" computation (Hammer et al., 2014, 2015).

By reifying the partial order of computation dependencies in a *demanded computation graph*

(DCG), this line of work provides a powerful and general approach to the design and implementation of efficient interactive systems. However, its low-level primitives make it difficult to express the complex fixed-point computations over cyclic control-flow graphs and recursive call structures that are found in arbitrary abstract interpretations (Stein et al., 2021a).

This dissertation takes a great deal of inspiration from demanded computation graphs, but specializes the language of demanded computations to abstract interpretation, both with syntactic structures and with query and edit semantics that dynamically modify the dependency graph to model program analysis computation.

3.3 Incremental & Demand-Driven Program Analysis

Although this dissertation (and corresponding conference papers) presents the first framework that is incremental, demand-driven *and* fully general with respect to abstract domain, there exist many techniques with some subset of those features, as shown in Table 3.1. This section discusses prior approaches to incremental and/or demand-driven program analysis, and includes some descriptions from Stein et al. (2021a).

The first row of Table 3.1 represents classic abstract interpretation frameworks, as described in Section 3.2.1. Although they support arbitrary abstract domains — notably including infinite-height domains with widening operators — they are neither incremental nor demand-driven.

Infinite-height domains with widening are of particular interest because they induce unbounded analysis computations; when a domain has finite height k, dataflow analysis requires at most $k \cdot |P|$ steps to converge on a program P. Although incremental and/or demand-driven dataflow analysis in finite-height domains is an important and interesting problem in its own right, it is unable to handle important analysis problems including most shape analyses and numerical analyses. Throughout this section, we use "dataflow analysis" to refer to the limited class of abstract interpretations in finite-height domains, such as the traditional monotone framework and reachability-based IFDS/IDE analyses (Kam and Ullman, 1976; Kildall, 1973; Reps et al., 1995; Sagiv et al., 1996).

3.3.1 Incremental Analysis

Incremental variants of many standard compiler analyses have been studied and developed in order to support responsive continuous compilation and structured editors/integrated development environments. These include both the aforementioned dataflow analyses (Ryder, 1983; Zadeck, 1984; Carroll and Ryder, 1988; Pollock and Soffa, 1989) and attribute grammars (Demers et al., 1981; Reps, 1982; Reps et al., 1983; Söderberg and Hedin, 2012), which combine parse trees with semantic information and can encode many analyses including dataflow.

Recent work has also contributed incremental versions of several broader classes of program analysis, including IFDS/IDE dataflow analyses (Arzt and Bodden, 2014; Do et al., 2017) and analysis DSLs based on extensions to Datalog (Szabó et al., 2016, 2018, 2021) These specialized approaches offer very effective solutions for these particular classes of program analysis, but place restrictions on abstract domains that rule out arbitrary abstract interpretations in infinite-height domains, for example by requiring domains to fall in the IFDS/IDE subset or by requiring that all domain operations are monotonic. These approaches are automatic (in that they require no user-provided specifications or loop invariants) and sound, but make no guarantee of from-scratch consistency and in fact violate it in some cases in order to maximize incremental reuse. On the other hand, Leino and Wüstholz (2015) propose a fine-grained incremental verification technique for the Boogie language, which verifies user-provided specifications of imperative procedures. Since these specifications include loop invariants, their algorithm can avoid altogether the issues and complexities introduced by cyclic dependencies.

3.3.2 Demand-Driven Analysis

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

Technique	Infinite domains with widening	Incremental	Demand-Driven
Classic Abstract Interpretation (Cousot and Cousot, 1977; Bourdoncle, 1993)	✓	✗	✗
Incremental Dataflow Analysis (Arzt and Bodden, 2014; Do et al., 2017; Ryder, 1983)	✗	✓	✗
Demand-Driven Dataflow Analysis (Reps, 1994; Duesterwald et al., 1995; Sagiv et al., 1996)	✗	✗	✓
Attribute Grammars (Magnusson and Hedin, 2007; Söderberg and Hedin, 2012)	✗	✓	✓
Datalog-Based Analyses (Szabó et al., 2016, 2018; Madsen et al., 2016)	✗	✓	✓
This dissertation	✓	✓	✓

Table 3.1: Feature comparison of existing analysis techniques and this dissertation. Citations are given as examples of the referenced categories of analysis techniques only, and are not intended to be exhaustive. The expressivity and generality of abstract interpretation makes it widely applicable and useful, but all existing approaches to incremental and/or demand-driven analysis of which we are aware restrict domain expressivity in some way. The goal of this dissertation is to define a framework for incremental and demand-driven analysis that is as general with respect to abstract domains as classic abstract interpretation.

elimination, parallel communication optimization, and integration testing (Bodík et al., 2000; Yuan et al., 1997; Duesterwald et al., 1996).

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

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

3.4 Interprocedural Analysis

Interprocedural dataflow analysis is a fundamentally more difficult and complex problem than intraprocedural, requiring analysis designers to strike a delicate balance between precise abstraction of control flow and exponential state explosion.

One popular approach is context sensitivity, which indexes program locations by some abstraction of calling context. For example, the "call strings" approach of Sharir and Pnueli (1981) and related modern approaches known colloquially as k-CFA or more precisely as k-call-site-sensitivity (Might et al., 2010) abstract concrete call stacks by truncation, considering dataflow into

each function separately for each k-length sequence of callers. Related approaches have been applied successfully to analyze object-oriented and dynamic imperative programming languages (Milanova et al., 2005; Andreasen and Møller, 2014), and the large body of work on control-flow analysis in functional languages is also closely related (Shivers, 1988; Might et al., 2010; Horn and Might, 2010).

Another class of interprocedural analysis techniques can be seen as variants of the "functional" approach of Sharir and Pnueli (1981), and are typically referred to as "compositional" or "modular". Interprocedural analyses in this style compute analysis facts for individual procedures, files, and/or compilation units, which are then composed in some way to yield some whole-program analysis results.

There is a great deal of ambiguity in the literature as to what it means for an analysis to be "compositional" or "modular"; in this dissertation, we draw a distinction between compositionality and modularity as follows.

A "compositional" analysis is any analysis that computes *composable* summaries of program parts' semantics, such that those results can be reused in different contexts. This includes, for example, tabulation algorithms for dataflow analysis and abstract interpretation (Sharir and Pnueli, 1981; Reps et al., 1995), bi-abduction (Calcagno and Distefano, 2011), and approaches based on reduction to finite-state machines (Chaki et al., 2004).

"Modularity" is a stronger form of compositionality: modular analyses are compositional analyses that analyze each program part *exactly once*, combining partial results from the bottom-up to produce whole-program results. In general, this requires either sophisticated special-purpose abstract domains that abstract relations over program states rather than program states themselves (Jeannet et al., 2010; Cousot and Cousot, 2002; Yorsh et al., 2008), or user-provided specifications of procedure pre-/post-conditions (Fähndrich and Logozzo, 2010).

Furthermore, although modularity naturally provides a degree of incrementality — in the sense that results need only be recomputed for changed program parts and their dependencies — this effect is often overstated for non-modular compositional analyses (e.g. in (Calcagno et al., 2011)). For example, the tabulation algorithm for interprocedural dataflow analysis is compositional

but not incremental (Arzt and Bodden, 2014).

Compositionality and modularity have both shown to be very effective for scaling program analyses to massive codebases in continuous integration pipelines (Calcagno and Distefano, 2011; Distefano et al., 2019; Fähndrich and Logozzo, 2010). Though these approaches effectively scale to industrial codebases, they are not intended to achieve real-time interactivity during the development process and apply only a very coarse-grained dependency tracking. In contrast, our aim is to support fine-grained incremental and demand-driven analysis in arbitrary abstract domains.

Chapter 4

Overview: Incremental & Demand-Driven Analysis with Dependency Graphs

In this section, we explain our technique and its applications at a high level, using some challenging code examples to demonstrate the obstacles faced by incremental and demand-driven abstract interpretation and show how they are addressed in this work.

We first address the problem of *intra*-procedural abstract interpretation in Section 4.1, describing our approach to incremental and demand-driven analysis in an imperative language without procedure calls. In particular, we show in Section 4.1.3 how fixed point computations with widening are out of reach of existing techniques and how dynamically updating analysis dependencies can bridge the gap. Portions of this section are drawn from our PLDI '21 paper "Demanded Abstract Interpretation" (Stein et al., 2021a).

Then, we describe and discuss our approach to the additional difficulties presented by *inter*-procedural abstract interpretation in the presence of general recursion in Section 4.2, i.e. analysis of a more realistic imperative programming language like C or Java. Portions of this section are drawn from a paper currently under submission: "Interactive Abstract Interpretation with Demanded Summarization", co-authored with David Flores, Bor-Yuh Evan Chang, and Manu Sridharan.

4.1 Intraprocedural Demanded Analysis

Consider a simple imperative program that appends two linked lists, as shown in Fig. 4.1.

Given well-formed (i.e., null-terminated and acyclic) input lists `p` and `q`, the `append` procedure must return a similarly well-formed list and not dereference **`null`** in order to be correct and

memory-safe.

These properties can be verified by a "shape analysis", consisting of abstract interpretation in a separation logic-based abstract domain (Berdine et al., 2005; Distefano et al., 2006; Magill et al., 2006). Abstract states in this domain are separation logic formulae, tracking facts like $\texttt{lseg}(p, null)$[1] to represent well-formedness of list p and using the separating conjunction $*$ to reason about disjoint regions of the heap.

Our goal is to enable interactive performance for arbitrary abstract interpretations, including such analyses, in respoonse to a user's edits and queries.

Some previous frameworks focus on either incremental (e.g., (Szabó et al., 2016; Arzt and Bodden, 2014)) or demand-driven static analysis (e.g., (Horwitz et al., 1995)), but they are unsuitable for this example since they either require analyses to be expressed in a DSL (precluding the use of existing highly-tuned implementations of complex domains) or are limited to finite domains (precluding infinite-height abstract domains, including this shape domain and most numerical domains).

The key challenge addressed by our approach to intraprocedural analysis is in handling of loops like the one at ℓ_3. In standard approaches to incremental computation, dependencies between entities are tracked in a directed graph, with graph reachability determining where re-computation is required after a modification. Crucially, this dependence graph is typically required to be *acyclic*. Unfortunately, abstract interpretation of loops naturally introduce cyclic dependencies, and for examples like the list-append program of Fig. 4.1 there is no clear syntactic method for removing the cycle (as the number of loop iterations depends on the data structures' size).

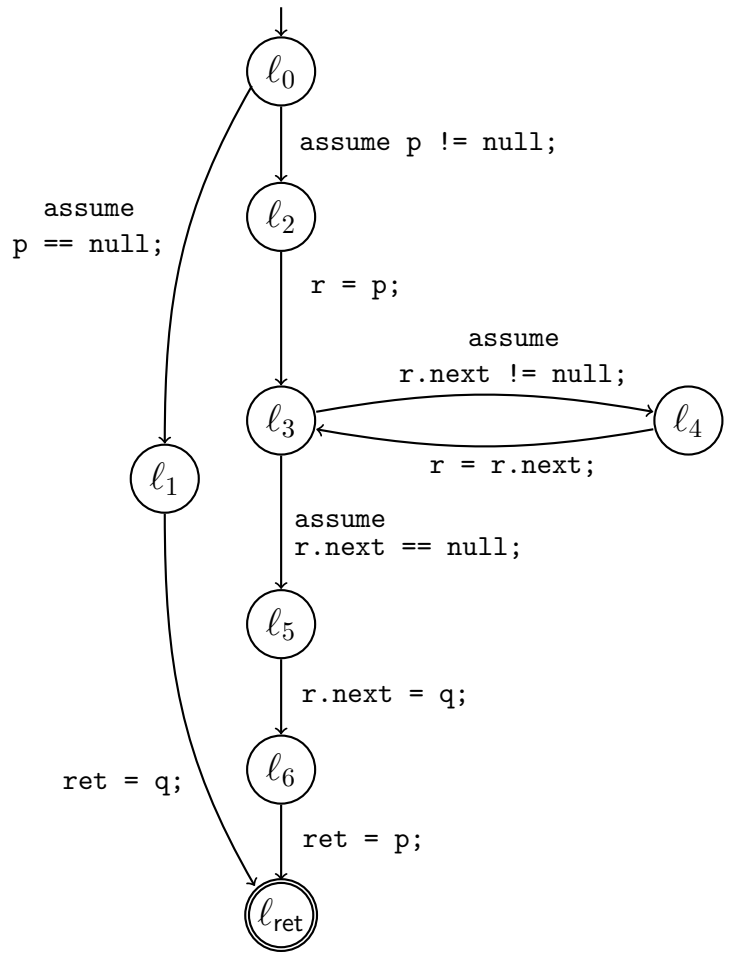
In this section, we illustrate our approach to *demanded*[2] abstract interpretation by example on this example program. We demonstrate how abstract interpretation of the program's control flow is reified in a *demanded abstract interpretation graph* or "DAIG" (Section 4.1.1), and then show how DAIGs support both demand-driven and incremental interactions with the underlying abstract

[1] i.e. "list segment" from address p to `null`

[2] We borrow the terminology "demanded" as short-hand for "incremental and demand-driven" from the *demanded computation graphs* of Hammer et al. (2014)

```
   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 }
```

(a) An imperative procedure to append two linked lists `p` and `q`. The labels ℓ_i mark program locations in its control flow.



(b) The control-flow graph (CFG) of the linked-list `append` procedure from Fig. 4.1. Vertices are program locations and edges are labelled by program statements. We encode structured conditional control-flow (`if`, `while`, etc.) into an unstructured control-flow graph in the standard way, using blocking `assume` statements and non-deterministic choice.

Figure 4.1: Two equivalent representations of an imperative linked list append procedure: (a) concrete syntax and (b) control-flow graph.

interpretation (Section 4.1.2). We highlight the key difficulties introduced by cyclic control flow and show how analysis thereof can be encoded acyclically and then extended dynamically to soundly compute fixed points in infinite-height abstract domains with widening (Section 4.1.3).

4.1.1 Reifying Abstract Interpretation in DAIGs

The `append` procedure from Fig. 4.1a may equivalently be represented as a control-flow graph (CFG) as shown in Fig. 4.1b, with vertices for program locations and edges labelled by atomic program statements.[3] A classical abstract interpreter analyzes such a program by starting with the initial abstract state at the entry location, then 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. 4.2 reifies the computational structure of such an abstract interpretation of the Fig. 4.1b CFG. Its vertices are uniquely-named mutable reference cells containing program syntax or abstract state, 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 analysis results. Reference cells for the fixed point state at a program location are named by the location (e.g., the name $\underline{\ell_0}$ names the reference cell for the fixed-point state φ_0 at program location ℓ_0). In Fig. 4.2 and throughout this dissertation, underlined symbols denote a name derived from that symbol: hashes, essentially.

To encode abstract interpretation computations, DAIG edges[4] are labelled by a symbol for an abstract interpretation semantic function and connect cells stores the function inputs to the cell storing the output, thus capturing the dependency structure of the program analysis. For example, the computation of the abstract transfer function over the CFG edge $\ell_0 \to \ell_1$ is encoded in Fig. 4.2 as a DAIG edge labelled by the transfer function symbol $[\![\cdot]\!]^\sharp$ from input cells $\underline{\ell_0}$ and $\underline{\ell_0} \cdot \underline{\ell_1}$

[3] 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.

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

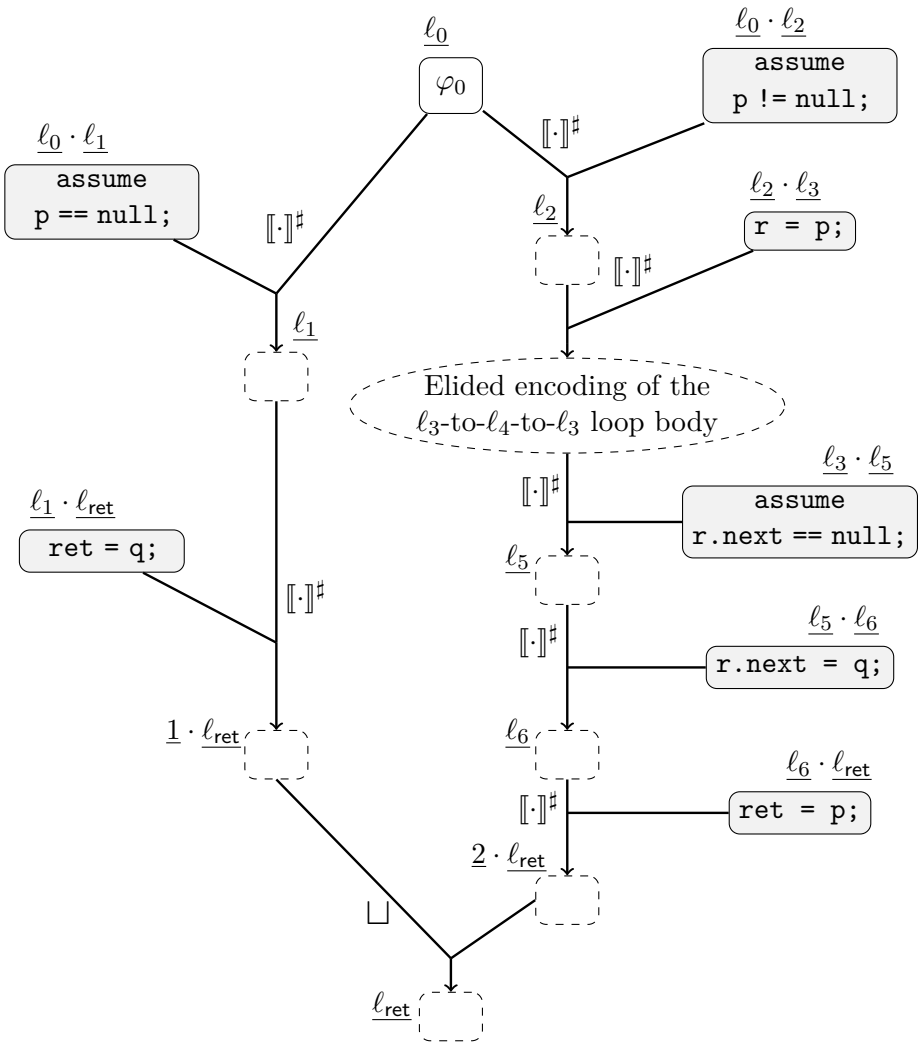


Figure 4.2: A demanded abstract interpretation graph (DAIG) for the program given in Fig. 4.1a before any queries are issued. The elided loop encoding is shown in Fig. 4.5.

(respectively containing the abstract state at ℓ_0 and the statement `assume p == null`) to output cell $\underline{\ell_1}$.

For control-flow join points (i.e., CFG nodes with multiple predecessors), the intermediate abstract states that feed into the join operation are stored in reference cells that are additionally indexed. For example, $\underline{1} \cdot \underline{\ell_{\mathsf{ret}}}$ and $\underline{2} \cdot \underline{\ell_{\mathsf{ret}}}$ name the cells for the abstract states to join to produce the abstract state to store in the cell named by $\underline{\ell_{\mathsf{ret}}}$, and the join computation is represented by an edge between these cells labelled by the join function symbol $\sqcup$.

Note that this presentation is agnostic to the acutal abstract domain chosen — although we use a separation logic-based shape domain for concreteness' sake in this section, we make no assumptions about its inner workings and effectively treat it as an existentially-typed module that exposes a minimal interface of transfer function $[\![\cdot]\!]^\sharp$, join $\sqcup$, widen ∇, a partial order $\sqsubseteq$, and an initial abstract state φ_0.

4.1.2 Demand-Driven and Incremental Analysis

The DAIG encoding naturally supports demand-driven and incremental analysis. When a reference cell is modified, forwards-reachable DAIG cells are *eagerly* marked as invalid since their contents are out-of-date and potentially inconsistent; these inconsistencies are then *lazily* address on demand.

This interplay between *eager* invalidation and *lazy* recomputation is key to the efficacy of demanded computation graph-based incremental computation (Hammer et al., 2014), maximizing the sound reuse of intermediate results while minimizing unnecessary computation of unneeded results.

We now proceed by example using 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.[5] This domain is of infinite height, has complex non-monotonic widening operators, and is absent a best abstraction function; it is therefore incompatible with previous

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

frameworks that are restricted to finite domains.

Demand-Driven Query Evaluation

In Fig. 4.3, we show the result of evaluating a demand query on our example DAIG. Suppose a client issues a query for the $\underline{1} \cdot \underline{\ell_{\mathsf{ret}}}$ cell, which corresponds to the abstract state at the **`return`** `q` statement at ℓ_1 in Fig. 4.1a. 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 $\underline{1} \cdot \underline{\ell_{\mathsf{ret}}}$, as shown in Fig. 4.3. Note that DAIGs are always acyclic, so this recursive traversal of dependencies is well-founded in the general case.

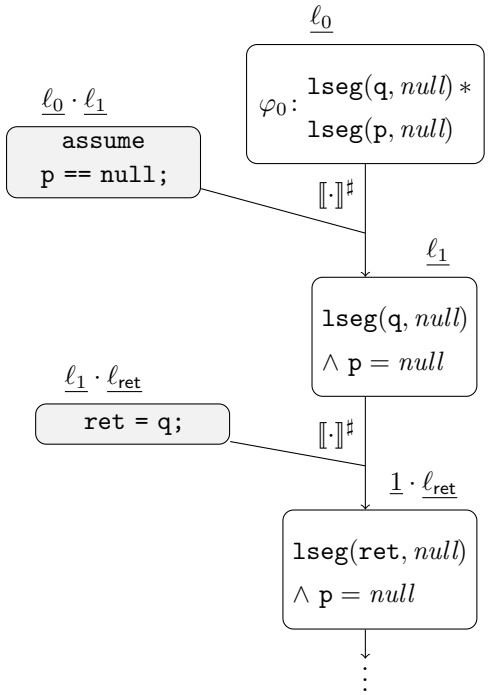


Figure 4.3: Demanding a value for $\underline{1} \cdot \underline{\ell_{\mathsf{ret}}}$ recursively triggers demand for its dependencies and is resolved by computing its value from the statements in $\underline{\ell_0} \cdot \underline{\ell_1}$ and $\underline{\ell_1} \cdot \underline{\ell_{\mathsf{ret}}}$ 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. 4.2.

Crucially, these results are now *memoized* for future reuse; a subsequent query for $\underline{\ell_{\mathrm{ret}}}$, 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.

Incremental Edits

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 (Hammer et al., 2015).

Consider a program edit which adds a logging statement `print("p is null")` just before the **`return`** at ℓ_1 in Fig. 4.1a. Intuitively, program behaviors are unchanged at those locations unreachable from the added statement, so an incremental analysis should only need to reanalyze the sub-DAIG reachable from the new statement.

Fig. 4.4 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, the cells $\underline{\ell_1} \cdot \underline{\ell_{\mathsf{ret}}}$ and $\underline{\ell_{\mathsf{ret}}}$ (containing abstract states φ' and φ''' respectively) are dirtied.

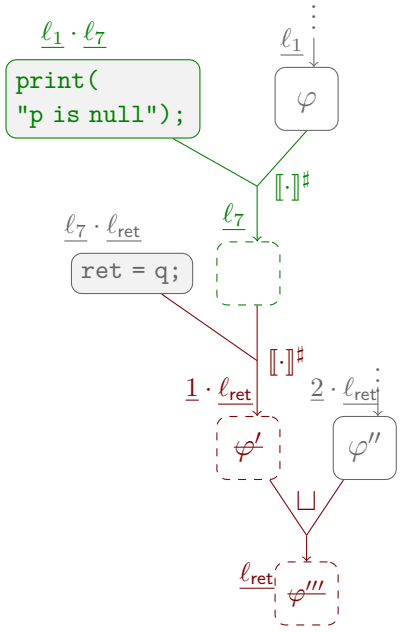


Figure 4.4: DAIG from Fig. 4.2 updated to reflect nodes added ($\underline{\ell_1} \cdot \underline{\ell_7}$ and $\underline{\ell_7}$) and potentially affected ($\underline{1} \cdot \underline{\ell_{\mathsf{ret}}}$ and $\underline{\ell_{\mathsf{ret}}}$) by the edit in Section 4.1.2. All other nodes are unchanged.

Note that, while nodes are dirtied *eagerly* in response to incremental edits, they are recomputed from up-to-date inputs *lazily*, only when demanded. That is, the DAIG encoding allows our analysis to avoid constant analysis recomputation as a program is edited. Instead, we compute analysis 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 $\underline{\ell_1}$ and $\underline{2} \cdot \underline{\ell_{\mathsf{ret}}}$ were both computed before the edit, a query for $\underline{\ell_{\mathrm{ret}}}$ must execute only two transfers and one join: the red and green edges of Fig. 4.4. 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 $\underline{[\![\cdot]\!]^\sharp} \cdot \underline{s_0} \cdot \underline{\varphi_0}$). 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.

4.1.3 Widening & Fixed Points

As shown in Fig. 4.2, encoding program structure and analysis data-flow into a demanded computation graph is relatively straightforward when the control-flow graph is acyclic. However, when handling cyclic control-flow structures like the loop at lines ℓ_3 and ℓ_4 of Fig. 4.1a, an abstract interpreter's fixed point computation is inherently cyclic. Properly handling cyclic control-flow and data-flow dependency structures is the crux of effective demand-driven and incremental abstract interpretation.

For the special case of data flow analysis in finite-height domains, this can be addressed by unrolling dependencies up to the height of the domain or similar. However, for arbitrary abstract interpretations there is no statically-fixed bound within which convergence is guaranteed, so static loop unrolling is insufficient to transform cyclic fixed point computations into acyclic dependency graphs.

On the other hand, introducing cyclic dependencies directly in the DAIG yields an unclear evaluation semantics for demand queries and incremental edits. The key insight we leverage is that we can instead enrich the semantics of demand-driven query evaluation and incremental edits to dynamically evolve DAIGs such that each step preserves the acyclic dependency structure invariant. To do so for loops, we use a distinguished DAIG 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 back by incremental edits. The details are formalized in Section 5.3; we proceed here by example, using the shape analysis of Fig. 4.1's `append` procedure to give the intuition of the technique.

Consider Fig. 4.5, and focus on the black and grey vertices and edges, ignoring the red for the

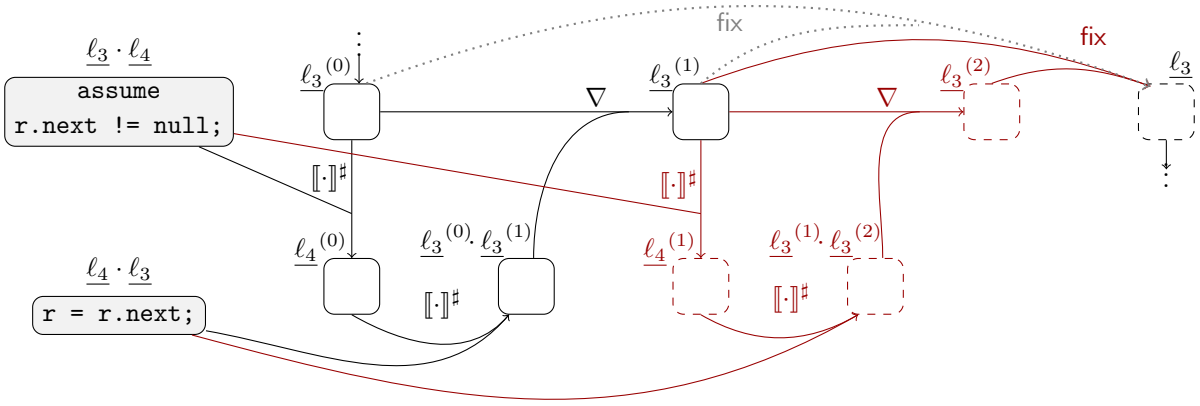


Figure 4.5: DAIG for the ℓ_3-to-ℓ_4-to-ℓ_3 loop of Fig. 4.1b after one demanded unrolling, with the new DAIG region shown in red and the (removed) pre-unrolling fix edge shown in dotted grey. Note that cells containing program syntax are not duplicated. The DAIG with the black vertices and edges along with the grey edge (but not the red ones) is the initial sub-DAIG for the dashed ellipse in Fig. 4.2.

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), connected by a DAIG encoding of the loop body with widening applied at the end. This representation corresponds to one iteration of the analysis fixed point computation over the loop. We also add a fix edge from these first two abstract iterates at the loop head ℓ_3 to a 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. 4.5, when the fixed-point $\underline{\ell_3}$ is demanded, its dependencies — the 0th and 1st abstract iterates $\underline{\ell_3}^{(0)}$ and $\underline{\ell_3}^{(1)}$ — are computed as described in the previous section. If their values are equal, then a fixed-point has been reached and may be written to $\underline{\ell_3}$.[6] 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 shown in the red cells and edges of Fig. 4.5. 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.

[6] We describe here the widening strategy of applying ∇ every iteration until a fixed-point is reached for simplicity, but the same general idea applies for other widening strategies or when checking convergence with $\sqsubseteq$ instead of $=$ to compute a pre-fixed-point.

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 edge, the fix edge is rolled back to a non-dirty cell (i.e., the 0th and 1st iterate). In Fig. 4.5, if the statement cell $\underline{\ell_4} \cdot \underline{\ell_3}$ 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.

4.2 Interprocedural Demanded Analysis

Although demanded unrolling effectively handles cyclic control flow within a single procedure (i.e. loops), it is insufficient for interprocedural analysis in the presence of general recursion.

In this section, we describe our approach to interprocedural demanded abstract interpretation by example on the recursive numerical program (and incremental edit thereof) shown in Fig. 4.6.

For illustration, we consider analysis of the original and edited programs using an *interval* abstract domain to bound variable ranges (Cousot and Cousot, 1977): a textbook example of an infinite-height domain with non-monotonic widening which is thus incompatible with many existing approaches to incremental and/or demand-driven analysis. In this domain, a sufficiently precise *inter*-procedural abstract interpreter can prove that $x = -1$ at the exit of `main` originally, and $x = -5$ at exit after the edit. Our key goal is to do so in a manner that is at once incremental, demand-driven and domain-agnostic.

4.2.1 Two Approaches to Interprocedural Analysis

Interprocedural abstract interpretation, if it is to be at all efficient or precise, requires some mechanism to jointly analyze similar calls to a procedure while keeping dissimilar calls separate, since procedures can in general be called in infinitely many contexts.

There are two classical approaches to this problem for batch analyses, referred to by Sharir and Pnueli (1981) as the "call strings" approach and the "functional" approach. Following Jeannet et al. (2010), we generalize this terminology and refer to the two approaches as *operational* and *denotational* respectively, since the call string approach (and related context-sensitive approaches, e.g. (Milanova et al., 2005; Might et al., 2010; Andreasen and Møller, 2014)) abstracts calling contexts directly and propagates analysis facts *operationally* along valid paths until a fixed point is reached, while the functional approach abstracts over procedure entry states to compute a *denotational* semantics for each procedure: an abstract transformer (as in e.g. (Cousot and Cousot, 2002; Jeannet et al., 2010; Yorsh et al., 2008)) or Hoare triples/summaries (as in e.g. (Sharir and Pnueli, 1981; Reps et al., 1995; Calcagno et al., 2011)) which can then be applied at callsites when the caller abstract state is compatible.

In this section, we describe the two techniques by worked example on the program shown in Fig. 4.6 and demonstrate that, although both are feasible in a batch setting, the choice between the

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

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

Figure 4.6: A simple numerical program written in an imperative language with recursive procedures, adapted from Sagiv et al. (1996), along with an edit applied at line 3. An incremental analysis should re-use many intermediate results from analysis of the original program when re-analyzing the edited version.

two has major and somewhat subtle ramifications on demanded analysis behavior.

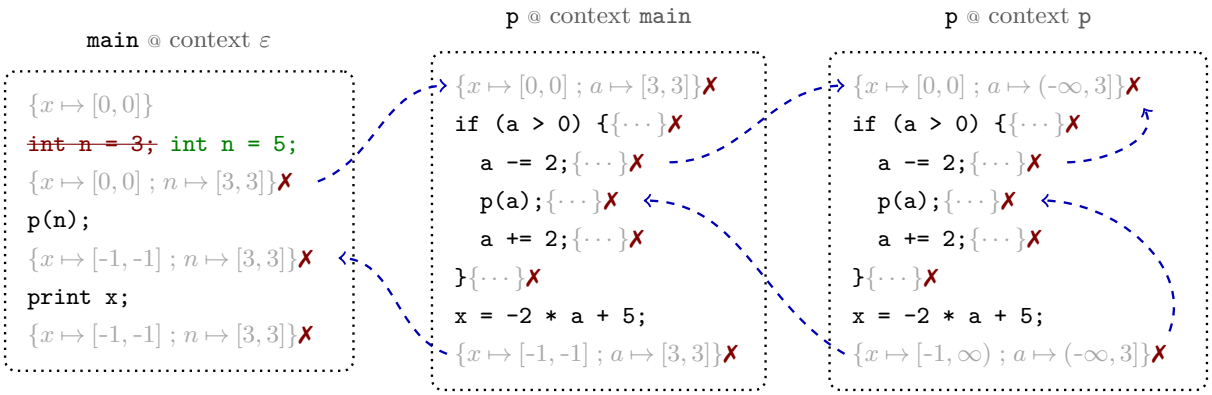
For batch analysis, both the operational approach and the denotational approach are quite effective. The abstract states computed in Fig. 4.7a and Fig. 4.7b are identical, and in both cases, `main` is analyzed once and `p` analyzed twice, with a fixed-point computation handling the recursive control flow in the second copy of `p`.

This is also borne out in practice and at scale: although there are tradeoffs between the two, both the denotational and the operational approach have seen widespread adoption in practical analysis tools. The key difference is that the operational approach distinguishes copies of a procedure by an abstraction of the concrete stack, whereas the denotational approach uses an abstraction of procedure-entry state.

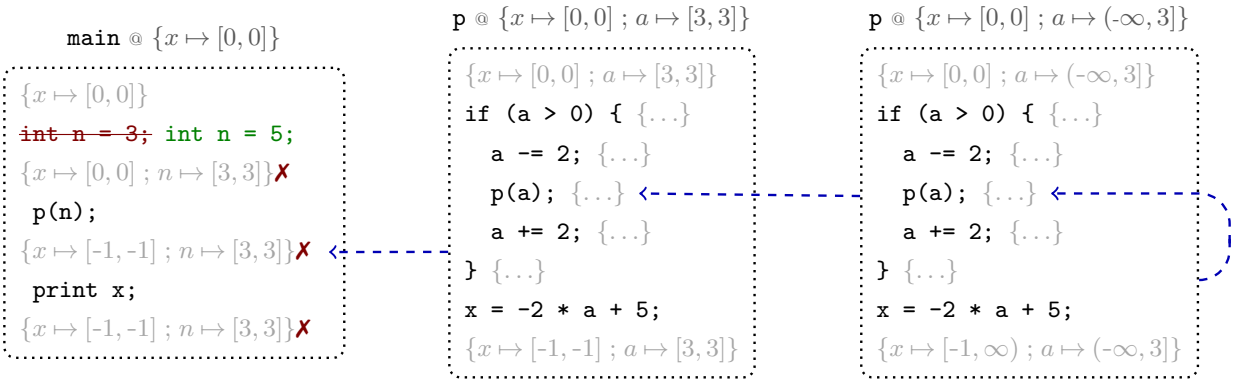
Thus, although the abstract states of Fig. 4.7a and Fig. 4.7b are rendered similarly here, they have markedly different meanings. Under the operational approach, an abstract state φ at a program location ℓ indexed by context c is interpreted as "φ abstracts all concrete states that can be reached at ℓ with a stack modelling c"; under the denotational approach, an abstract state φ at a location ℓ indexed by abstract state φ' is interpreted as a Hoare triple over the valid paths to ℓ from the entry of the containing procedure with precondition φ' and postcondition φ.

This distinction carries significant ramifications for analysis behavior in the presence of incremental edits and demand queries, as demonstrated by the impact of the program edit in Fig. 4.7.

Under the operational approach, all previously-computed analysis results reachable via intraprocedural dataflow, calls, or returns must be dirtied, as they may be affected by the upstream edit. Under the denotational approach, only those results reachable via intraprocedural dataflow and returns are rendered invalid. That is, although the edit in `main` may require the tabulation of different summaries of `p` to respond to future queries, it has no effect on the validity of tabulated summaries, so the analysis facts computed in the two `p` summaries can still be reused.



(a) Batch 1-call-string-sensitive interval analysis of the pre-edit program, typifying the *operational* approach. Procedure-entry abstract states depend on the abstract states at context-relevant call-sites, and return-site abstract states depend on the abstract states at context-relevant callee exits.



(b) Batch tabulation/summary-based interval analysis of the pre-edit program, typifying the *denotational* approach. Return-site abstract states depend on the summary applied at the corresponding procedure call.

Figure 4.7: Two approaches to batch interprocedural abstract interpretation of the pre-edit program given in Fig. 4.6. We can see that both approaches are similarly effective for batch analysis, yielding the same results and performing roughly the same amount of analysis computation. Those abstract states invalidated by the program edit are marked by a ✗, and blue dashed arrows denote interprocedural analysis dependencies.

4.2.2 Demanded Summarization

Our demanded abstract interpretation technique thus takes a denotational approach to interprocedural analysis, building on intraprocedural DAIGs with a technique that we call *demanded summarization*. At a high-level, demanded summarization works by instantiating DAIGs on demand to produce compositional, denotational summaries, drawing inspiration from tabulation-based approaches (Sharir and Pnueli, 1981; Reps et al., 1995).

DAIGs, as described in the previous section, are a powerful building block for incremental, demand-driven and domain-agnostic *intra*-procedural abstract interpretation, but they have no mechanism for dealing with procedure calls, recursive or otherwise. Furthermore, they intuitively reify an "operational" abstract semantics (i.e. how to interpret commands to compute the abstract state at a program point, given the abstract state at predecessor points).

The key technical challenge addressed is in applying denotational procedure summaries in the middle of an operational DAIG-based analysis, while supporting arbitrary abstract domains and recursive procedures and preserving the desired meta-theoretical guarantees of soundness, termination and from-scratch consistency.

As with intraprocedural analysis of loops, the technique is based on dynamic tracking of dependencies, allowing for the reification of *a priori*-unbounded analysis computations into finite dependency structures amenable to demanded evaluation. We call this structure a *demanded summarization graph* (DSG), and it embodies three core ideas.

First, each denotational procedure summary is computed using a DAIG, which is a node in the DSG. Each individual DAIG is guaranteed to be from-scratch consistent for intraprocedural analysis (Stein et al., 2021a).

Second, at a procedure call, a DAIG summary is computed on demand (or re-used if already computed); an edge in the DSG tracks where the summary is applied. To ensure from-scratch consistency, a procedure summary is only applied at call sites *after* computation of that summary has converged.

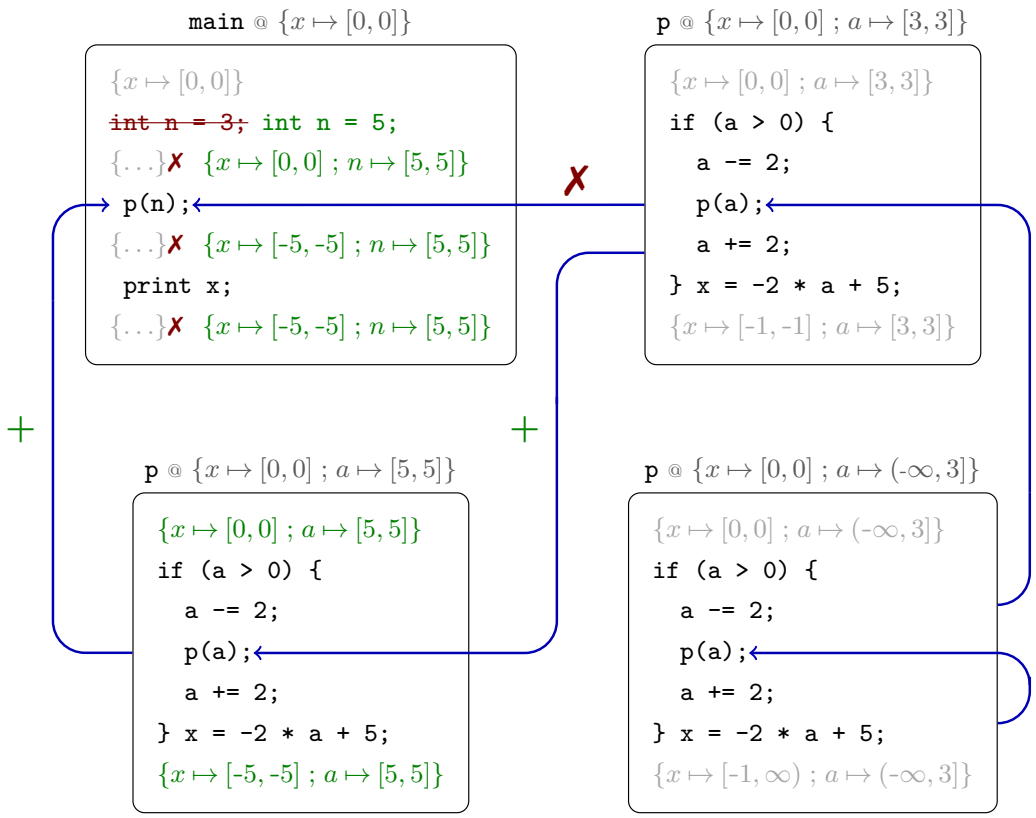


Figure 4.8: Demanded summarization applied to the program and edit of Fig. 4.6. After the edit at line 3, the previously-computed summaries of `p` (on the right half of this figure) are still valid and can be used to produce the additional `p` summary needed to reanalyze `main`. The summary dependency edge labelled by a ✗ is removed when the callsite it points to is dirtied by the edit. When a new query is issued at the exit of `main`, only the green states must be recomputed, instantiating a new partial summary of `p` and two new dependency edges (labelled by +) in the process.

Third, recursion is handled via an additional fixed point computation *within* the recursive procedure's DAIG, designed carefully to maintain from-scratch consistency.

Fig. 4.8 shows how the running example is analyzed using DSGs. Assume an initial query for the analysis state at the exit of `main` when $x = 0$ at `main`'s entry. We first create a corresponding DAIG (labelled `main` @ $\{x \mapsto [0, 0]\}$) to perform this analysis. Intraprocedural analysis proceeds until reaching the call `p(n)` with fact $\{x \mapsto [0, 0]; n \mapsto [3, 3]\}$. Handling the call requires computing a summary for `p` by instantiating a new DAIG for `p` (at top right of Fig. 4.8).

Recursion. Analysis within the DAIG for `p` with fact $\{x \mapsto [0, 0]; n \mapsto [3, 3]\}$ reaches a recursive call `p(a)`. Here, the DSG approach recognizes the call as recursive and applies widening before instantiating another DAIG over `p`, yielding initial state $\{x \mapsto [0, 0]; a \mapsto (-\infty, 3]\}$ (at bottom right of Fig. 4.8). Analysis in this new DAIG once again reaches the recursive callsite, but widening of the entry state has converged to $\{x \mapsto [0, 0]; a \mapsto (-\infty, 3]\}$.

This process of creating fresh DAIGs for recursive calls until the entry state converges is analogous to the demanded loop unrolling technique used for DAIGs and described in Section 4.1.3. But now, we must analyze the control-flow path(s) *after* the recursive call in the final (self-dependent) DAIG, as we need a converged summary before we can propagate back to callers. Further, a fixed point may be required to converge on the exit state.

To obtain a converged exit state, we run a carefully-designed fixed-point analysis *within* the final DAIG, using intraprocedural dirtying to iterate. The process is illustrated in Fig. 4.9. It starts by injecting $\bot$ as the post-state of the final recursive call (step (a)), since no dataflow has yet reached the exit. Propagating to the exit yields a fact $\{x \mapsto [5, \infty]; a \mapsto (-\infty, 0]\}$, due to the join with the non-recursive path. To iterate, we dirty the post-state of the recursive call within the DAIG, update the post-state to this new fact, and then re-query the exit state (step (b)). We continue this process one more time (step (c)), and see the exit state converges to $\{x \mapsto [-1, \infty]; a \mapsto (-\infty, 3]\}$.

Finally, the fully-analyzed (bottom-right) DAIG of Fig. 4.9 can be applied as a summary in the DAIG that initially demanded it (at top right of Fig. 4.8), which can then be fully analyzed and applied as a summary in the `main` DAIG. We add interprocedural dependency edges to the DSG to

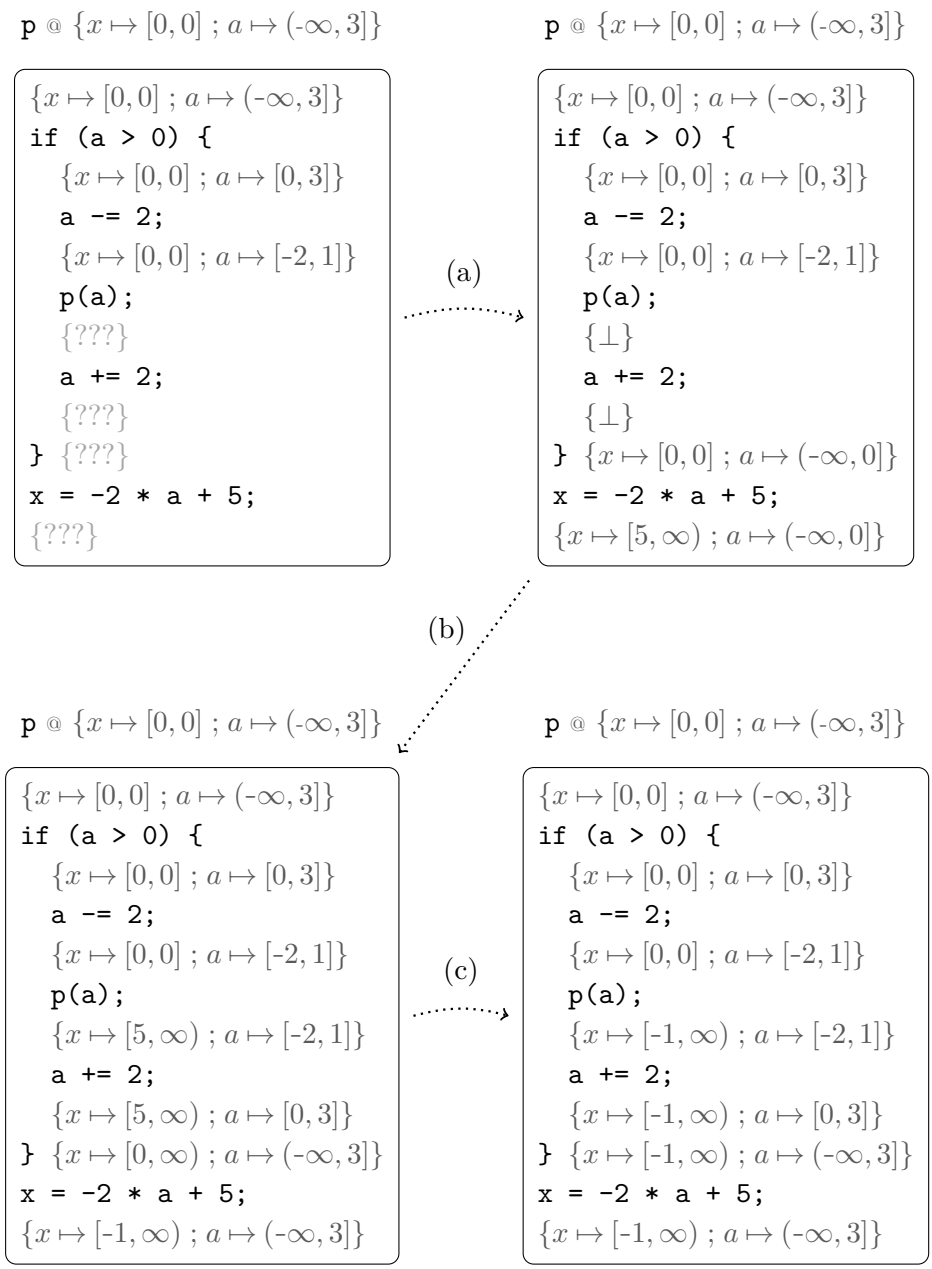


Figure 4.9: In-place fixed point computation of the self-referential summary of Fig. 4.8, showing how analysis converges on the control-flow cycle between its recursive return site and procedure exit location.

track each of these summary applications: the blue edges not marked by a + in Fig. 4.8. Finally, we finish analyzing `main` and return the query result $\{x \mapsto [-1,-1]; n \mapsto [3,3]\}$.

Incremental Updates. Now, when the edit is made to `main`, its downstream dependencies (the states marked by ✗ in Fig. 4.8) are dirtied in the `main` DAIG, but no other DAIG is affected because the `main` summary has no dependencies. Suppose another query is issued for the exit abstract state of `main`: we now reach the `p(n)` callsite with no applicable summary and so instantiate a new DAIG for `p` with initial state $\{x \mapsto [0,0]; a \mapsto [5,5]\}$, at bottom left of Fig. 4.8. However, the previously-computed summary with precondition $\{x \mapsto [0,0]; a \mapsto [3,3]\}$ can now be used at the recursive callsite `p(a)`, allowing analysis to complete without recomputing a fixed point over the recursive procedure. Once again, summary applications are tracked by dependency edges to enable precise dirtying in response to future edits.

Note that this summary is actually more precise than the one that would have been computed by from-scratch batch analysis of the updated program. A DSG may apply a compatible summary whose precondition is stronger than the result of widening at recursive callsites without loss of precision, though its result may be more precise than from-scratch batch analysis if it does so — a technical violation of from-scratch consistency that we see as harmless and potentially beneficial.

Demanded Summarization and Compositional Analysis. When a new summary is required to analyze a procedure call, demanded summarization instantiates a new DAIG with the requisite initial abstract state and issues a query for the abstract state at its exit. Within an instantiated DAIG, we can reuse results to respond to future queries and efficiently dirty results in response to program edits. However, instantiated DAIGs cost memory, so a balance must be struck between the granularity of cached analysis results and memory usage. We can tune this balance by condensing DAIGs to two-state summaries, remembering the entry and exit states — essentially a Hoare triple over the procedure — while discarding all intermediate analysis facts that contributed to said triple.

The flip side is also useful: a demanded summarization-based analysis can be initialized with procedure summaries computed from a batch compositional analysis, provided that dependencies between summaries are preserved to enable dirtying after edits. This addresses the motivating

problem of connecting server-based compositional analysis with interactive analysis in the IDE.

Chapter 5

Intraprocedural Analysis with Demanded Abstract Interpretation Graphs

In this chapter, we present a framework for incremental and demand-driven intraprocedural abstract interpretation in arbitrary domains. The presentation begins with some necessary background, giving definitions and fixing syntax for programs, their concrete semantics, and batch abstract interpretation. Then, we define *demanded abstract interpretation graphs* (DAIGs), which reify the abstract interpretation of a program as a graph structure that explicitly represents program statements, abstract states, and the dependency structure of analysis computations.

Next, we specify an operational semantics for DAIGs that realizes incremental updates and demand-driven evaluation via demanded unrolling of abstract interpretation fixed-point copmutations, then prove that it preserves soundness and termination, and that its results are from-scratch consistent with classical batch abstract interpretation by global fixed-point iteration.

Lastly, experimental evaluation of a prototype implementation provides evidence for the expressivity and efficacy of demanded abstract interpretation, instantiating the analysis with interval, octagon, and shape domains and producing analysis results at real-time interactive speeds.

This chapter includes portions of the PLDI 2021 paper "Demanded Abstract Interpretation" (Stein et al., 2021a), which was co-authored with Bor-Yuh Evan Chang and Manu Sridharan.

5.1 Program Syntax and Concrete Semantics

This chapter describes how to lift a program and an an abstract interpreter *together* into a demanded abstract interpretation graph (DAIG), a representation that is amenable to sound

incremental and demand-driven 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 give a generic imperative programming language and its concrete semantics, so as to fix syntax, terminology, and definitions for the sections to follow.

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

Figure 5.1: A generic programming language of control-flow graphs edge-labelled by an unspecified statement language.

A program $P = \langle L, E, \ell_0 \rangle$ 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. Programs are represented as unstructured control-flow graphs (CFGs) thus so as to simplify presentation and provide genericty across different imperative languages and their various control-flow semantics. That is, rather than defining a fixed concrete syntax of structured control-flow constructs (such as `if`, `while`, etc.) we assume that a language frontend has already transformed the program into a CFG, as is nearly always done before any data flow analysis or abstract interpretation.

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[1] 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* $\mathsf{loop}(\ell') : \mathcal{P}(L)$, the set of locations from which ℓ can be reached without passing

[1] A vertex ℓ *dominates* a vertex ℓ' if every path from the entry ℓ_0 to ℓ' passes through ℓ.

through ℓ'. Intuitively, the natural loop is the "body" of the loop with head ℓ', and we define $\mathsf{loop}(\ell) \triangleq \emptyset$ 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., (Aho et al., 2006), Sec. 9.6) which we do not repeat here.

We also define a helper function $\mathsf{fwd\text{-}edges\text{-}to}_{\langle L,E,\ell_0\rangle} : \mathit{Loc} \to \mathcal{P}(\mathbb{N} \times \mathit{Edge})$ 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 $\{\ell \mid |\mathsf{fwd\text{-}edges\text{-}to}(\ell)| \geq 2\}$ 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.

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 (i.e. $\forall\ \ell \dashv[\, s \,]\mapsto \ell' \in E\ .\ \ell \in L \wedge \ell' \in L$)

(2) $\langle L, E, \ell_0\rangle$ forms a reducible control-flow graph.

These conditions ensure that we avoid degenerate edge cases and only consider control-flow graph which correspond to realistic programs (Aho et al., 2006).

Statement syntax is left fully unspecified, again so as to simplify presentation and remain as generic as possible. A statement $s \in \mathit{Stmt}$ is given meaning only by the concrete denotational semantics $[\![\cdot]\!]$, which interprets it as a partial function $[\![s]\!]$ over concrete program states $\sigma \in \Sigma$.

Such a concrete semantics of *statements* can also be lifted to a concrete semantics of *programs* as follows. We define the *collecting*[2] semantics $[\![\cdot]\!]^*_P : \mathit{Loc} \to \mathcal{P}(\Sigma)$ of a program $P = \langle L, E, \ell_0\rangle$ as the least fixed point of the following system of equations:

$$
\begin{aligned}
[\![\ \ell_0\ ]\!]^*_P &= \{\sigma_0\} \\
[\![\ \ell\ \ ]\!]^*_P &= \left\{\ [\![s]\!]\sigma \;\middle|\; \exists\ \ell' \dashv[\, s \,]\mapsto \ell \in E\ .\ \sigma \in [\![\ell']\!]^*_{\langle L,E,\ell_0\rangle}\right\}
\end{aligned}
$$

[2] So-called because it "collects" the concrete states reachable at each program location

That is, $[\![\ell]\!]^*_P$ is the set of all concrete states that can be witnessed at program location ℓ in a valid concrete execution of P, given by the transitive closure of the statement semantics over the program's control-flow graph. We elide the subscript P when it is clear from context. Such a collecting semantics is infinite and 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 intraprocedural 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*[3] $\sqsubseteq \ \in \mathcal{P}(\Sigma^\sharp \times \Sigma^\sharp)$ with a *bottom* element $\bot \in \Sigma^\sharp$ such that $\bot \sqsubseteq \varphi$ for all $\varphi \in \Sigma^\sharp$
  - an *upper bound* (a.k.a. *join*) operator $\sqcup \ : \ \Sigma^\sharp \to \Sigma^\sharp \to \Sigma^\sharp$ such that $\varphi_1 \sqsubseteq \varphi_1 \sqcup \varphi_2$ and $\varphi_2 \sqsubseteq \varphi_1 \sqcup \varphi_2$ for all φ_1, φ_2.
- An *initial abstract state* $\varphi_0 \in \Sigma^\sharp$ which abstracts the initial concrete state σ_0
- An *abstract semantics* $[\![\cdot]\!]^\sharp : \mathit{Stmt} \to \Sigma^\sharp \to \Sigma^\sharp$ that interprets program statements as functions over *abstract* states. Crucially, $[\![\cdot]\!]^\sharp$ must be monotone: for all s, if $\varphi \sqsubseteq \varphi'$ then $[\![s]\!]^\sharp \varphi \sqsubseteq [\![s]\!]^\sharp \varphi'$
- A *widening* operator $\nabla \ : \ \Sigma^\sharp \to \Sigma^\sharp \to \Sigma^\sharp$ that is an upper bound operator (as defined for join) and guarantees convergence by satisfying the *ascending chain condition*: 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. Note that, for abstract domains where the lattice $(\Sigma^\sharp, \sqsubseteq)$ has finite height, ∇ may be equal to $\sqcup$ since all increasing sequences converge.

[3] i.e. a reflexive, transitive, antisymmetric binary relation on abstract states

Furthermore, a concretization function $\gamma : \Sigma^\sharp \to \mathcal{P}(\Sigma)$ gives meaning to abstract states by relating them to concrete states. We say that a concrete state σ *models* an abstract state φ (equivalently, that φ *abstracts* σ), written $\sigma \models \varphi$, when $\sigma \in \gamma(\varphi)$. Note that, despite its usefulness in formalism, the concretization (γ) function is not included in the abstract interpreter tuple because it is not computable in general and therefore not used when implementing an abstract interpreter.

Definition 5.1 (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* $[\![s]\!]\sigma \neq \bot$ *then* $[\![s]\!]\sigma \models [\![s]\!]^\sharp\varphi$*.*

That is, an abstract interpreter is locally sound if its abstract transfer function $[\![\cdot]\!]$ conservatively over-approximates the concrete semantics for all statements s: if φ abstracts σ, then the abstract interpretation of s in φ must abstract the concrete interpretation of s in σ.

This local soundness property 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.

Theorem 5.1 (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* $[\![\cdot]\!]^{\sharp *}_{\langle L,E,\ell_0 \rangle} : Loc \to \Sigma^\sharp$ *such that for all* $\sigma \in [\![\ell]\!]^*_{\langle L,E,\ell_0 \rangle}$*,* $\sigma \models [\![\ell]\!]^{\sharp *}_{\langle L,E,\ell_0 \rangle}$*.*

Such an abstract collecting semantics can be defined analogously to the concrete semantics as 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 (Cousot and Cousot, 1977) and that such a global fixed-point is computable using the chaotic iteration method with widening (Bourdoncle, 1993).

5.2 Demanded Abstract Interpretation Graphs

Recall from Section 4.1 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 dataflow relationships among those cells.

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

In Fig. 5.2, 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 hypergraph is a generalization of a standard graph whose edges connect a set of sources to a set of destinations; in our construction, the destination set is always a singleton, since 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. An underlined syntactic object can be understood essentially as the hash of that object, so two such names are equal if the underlined terms are equal.

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. Equalities over these more-complex names 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.

The visual representation of DAIGs used in the figures of Section 4.1 is isomorphic to this

presentation: each box with contents v and label n represents a reference cell $n[v : \tau]$, where τ is $\Sigma^\sharp$ for un-shaded boxes and *Stmt* for shaded boxes; each dashed box represents an empty reference cell $n[\varepsilon : \Sigma^\sharp]$; and each edge with sources labelled $n_1, \ldots, n_k$, target labelled n, and label f represents a computation edge $n \leftarrow f(n_1, \ldots, n_k)$.

Throughout this chapter and the remainder of the dissertation, we will make use of some shorthands as follows. 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[4] as f's output.

Definition 5.2 (DAIG Well-formedness)**.** *A DAIG $\langle R, C \rangle$ is subject to the following well-formedness constraints.*

(1) References are named uniquely: $\forall r, r' \in R \;.\; \mathit{name}(r) = \mathit{name}(r') \;\Leftrightarrow\; r = r'$

(2) Computations have unique destinations: $\forall c, c' \in C \;.\; \mathit{dest}(c) = \mathit{dest}(c') \;\Leftrightarrow\; c = c'$

(3) Computations are well-typed with respect to references: $\forall c \in C \;.\; R \vdash c$

(4) Dependencies are acyclic: $\nexists\, r \in R \;.\; \mathit{name}(r) \rightsquigarrow \mathit{name}(r)$

(5) Empty references have dependencies: $\forall n[\varepsilon : \tau] \in R \;.\; \exists\, c \in C \;.\; n = \mathit{dest}(c)$

These constraints together ensure that demand-driven evaluation and incremental change propagation have unambiguous and consistent semantics. That is, in a well-formed DAIG the value of any empty cell can be computed on demand and the region affected by an edit can be identified and dirtied.

[4] The fix function — which triggers demanded unrolling to compute loop fixed points and will be described in detail in the following section — is typed as a binary operator on abstract states.

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 5.1), there are three general cases shown in Fig. 5.3 to consider when examining a corresponding DAIG.

Definition 5.3 (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. 5.3 for a visual representation of this definition to build intuition for the more formal statements below.

(1) Forward CFG edges $\ell' \dashv[\, s \,]\mapsto \ell$ 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).[5]

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

(2) Forward CFG edges to *join* locations are slightly more complex: for each join location ℓ, $\mathsf{fwd\text{-}edges\text{-}to}$ 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.

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

[5] 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.

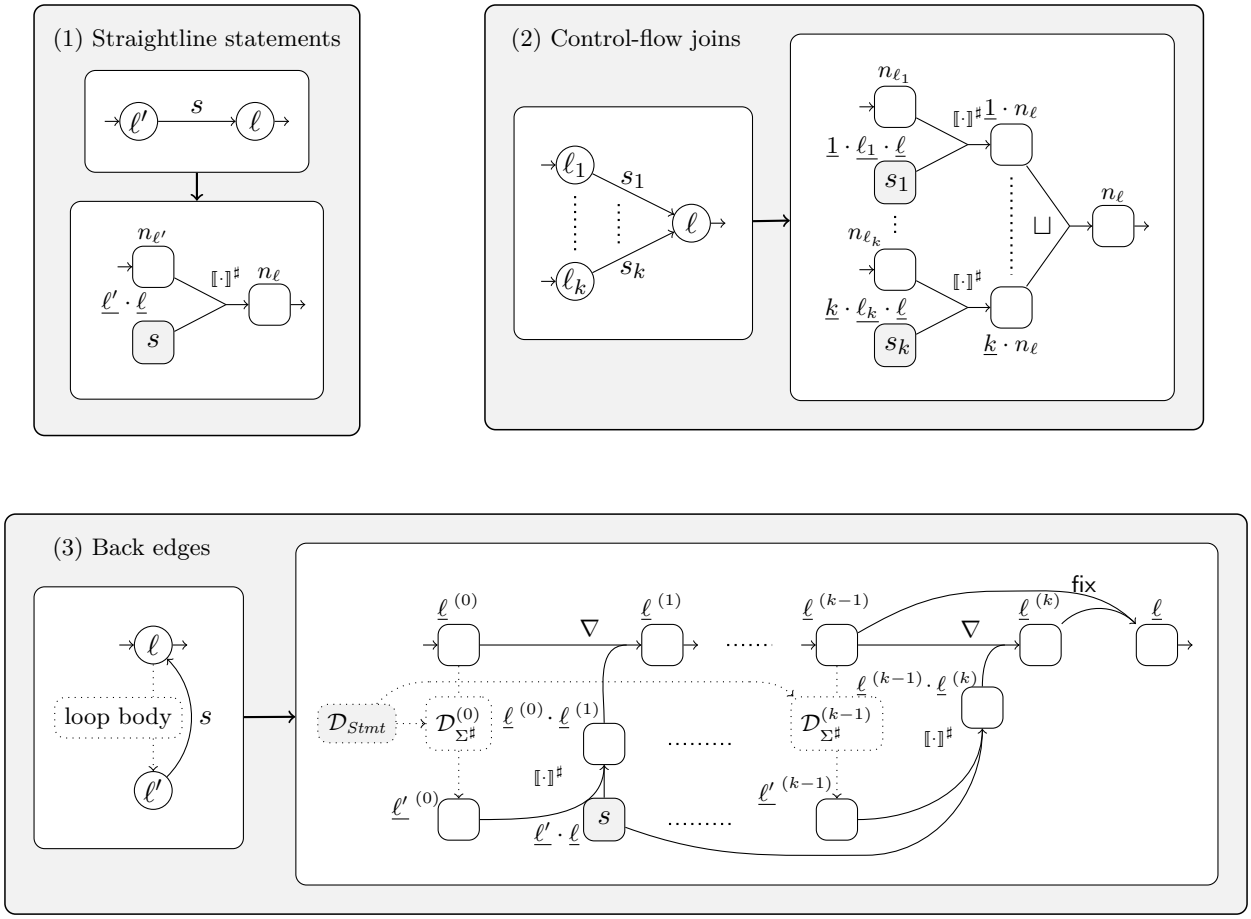


Figure 5.3: DAIG–CFG Consistency (Definition 5.3) in diagram form, illustrating how different CFG structures are encoded into DAIG structures. In subfigure (3), we apply some ad-hoc shorthands for the DAIG encoding of the loop body: $\mathcal{D}_{Stmt}$ contains all of its statement reference cells, 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.

(3) As described informally in Section 4.1.3, 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 $\ell' \dashv[\, s \,]\!\rightarrow \ell$, 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 [6]. 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 5.3), 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.

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

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 $\mathsf{loop\text{-}comps}(\ell)$ the subset of C whose elements contain a name $\underline{\ell'}^{(0)}$ such that $\ell' \in \mathsf{loop}(\ell)$. The helper function incr increments the loop-iteration counts in names, and we write $\mathsf{incr\text{-}c}$ for the pointwise lifting of incr to edges $c \in \mathit{Comp}$, i.e.

$$
\begin{aligned}
\mathsf{incr}(n) \quad &\triangleq \quad \begin{cases} \underline{\ell}^{(k+1)} & \text{if } n = \underline{\ell}^{(k)} \\ \mathsf{incr}(n_1) \cdot \mathsf{incr}(n_2) & \text{if } n = n_1 \cdot n_2 \\ n & \text{otherwise} \end{cases} \\
\mathsf{incr\text{-}c}\,(n \leftarrow f(n_1, \ldots, n_k)) \quad &\triangleq \quad \mathsf{incr}(n) \leftarrow f\,(\mathsf{incr}(n_1), \ldots \mathsf{incr}(n_k))
\end{aligned}
$$

The above establishes when the structure of a DAIG is consistent with a program CFG. A DAIG is consistent with an abstract interpreter when the partial analysis results stored in the DAIG

are consistent with the operations of the underlying abstract domain, or formally as follows:

Definition 5.4 (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 $\mathcal{D} \approxeq \langle \Sigma^\sharp, \varphi_0, [\![\cdot]\!]^\sharp, \sqsubseteq, \sqcup, \nabla\rangle$, 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*

$$\forall\ n[v : \Sigma^\sharp] \in R\ ,\quad n \leftarrow f(n_1, ..., n_k) \in C\quad .$$

$$\{n_i[v_i : \tau_i]\ \mid\ 1 \le i \le k\} \subseteq R\ \wedge\ \begin{cases} v = v_1 = v_2 & \text{if } f = \mathsf{fix} \\ v = f(v_1, \ldots, v_k) & \text{otherwise} \end{cases}$$

Together, the CFG-consistency property of Definition 5.3 and the AI-consistency property of Definition 5.4 ensure that a well-formed DAIG accurately encodes a partially-evaluated abstract interpretation state of the program. The key property that we will show in the following section is that both forms of consistency are preserved by both queries and edits, meaning that the abstract interpretation state of a DAIG can reliably and safely be updated during interactive use.

In addition to those *preservation* results, it is important to show that an initial DAIG which is consistent in both respects can be constructed for an arbitrary program CFG and abstract interpreter.

Definition 5.5 (Initial DAIG Construction)**.** *The initial DAIG $\mathcal{D}_{\mathsf{init}}(\langle L, E, \ell_0\rangle, \langle \Sigma^\sharp, \varphi_0, [\![\cdot]\!]^\sharp, \sqsubseteq, \sqcup, \nabla\rangle)$ 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 conceptually-distinct sets of both reference cells and computation edges, which we describe and define separately for clarity and ease of presentation.*

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

- 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[6]

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

control-flow edges share a destination.

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

- $R_{\Sigma^\sharp}$ 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.

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

- 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 $\mathsf{src\text{-}nm}(\ell, \ell')$ to be $\underline{\ell}$ when ℓ is a loop head and $\ell' \notin \mathsf{loop}(\ell)$ and n_ℓ otherwise.

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

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

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

- 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.

$$C_{\mathsf{fix}} \triangleq \left\{ \underline{\ell} \leftarrow \mathsf{fix}\left(\underline{\ell}^{(0)}, \underline{\ell}^{(1)}\right) \;\middle|\; \ell' \dashv[\, s \,]\mapsto \ell \in E_b \right\}$$

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

$$C_\nabla \triangleq \left\{ \underline{\ell}^{(1)} \leftarrow \nabla(\underline{\ell}^{(0)},\ \ \underline{\ell}^{(0)} \cdot\ \underline{\ell}^{(1)}) \;\middle|\; \ell' \dashv[\, s \,]\mapsto \ell \in E_b \right\}$$

We elide the abstract domain parameter to $\mathcal{D}_{\mathsf{init}}$ where it is irrelevant or clear from context.

Lemma 5.1 (Initial DAIG Well-Formedness, CFG-Consistency, and AI-Consistency)**.** *For all well-formed programs $\langle L, E, \ell_0\rangle$, $\mathcal{D}_{\mathit{init}}(\langle L, E, \ell_0\rangle)$ is well-formed, $\mathcal{D}_{\mathit{init}}(\langle L, E, \ell_0\rangle) \cong \langle L, E, \ell_0\rangle$, and $\mathcal{D}_{\mathit{init}}(\langle L, E, \ell_0\rangle) \cong \langle \Sigma^\sharp, \varphi_0, [\![\cdot]\!]^\sharp, \sqsubseteq, \sqcup, \nabla\rangle$.*

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.

(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, as it is straightforward to verify by cross-referencing the conditions of Definition 5.3 with the set constructions of Definition 5.5. 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. □

5.3 DAIG Semantics for Demanded Abstract Interpretation

This section 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, $\mathsf{dom}(M)$ 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. We also note that memoization tables M are invariably sound in the standard sense: $M(\underline{f} \cdot \underline{v_1} \cdots \underline{v_k}) = f(v_1, \ldots, v_k)$.

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.

5.3.1 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 by the $\mathcal{D}, M \vdash n \Rightarrow v \; ; \mathcal{D}', M'$ judgment form, whose inference rules are given in Fig. 5.4.

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

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

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

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

Figure 5.4: 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 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 $\mathcal{D}[n \mapsto v]$ notation denotes a low-level mutation of the reference cell named by n, not an external edit that would trigger invalidation (which we will describe below in Section 6.2.2).

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).

The Q-Loop-Converge and Q-Loop-Unroll rules govern the analysis of cyclic control flow and are are described in detail separately, in Section 5.3.2.

5.3.2 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 4.1.3 and represented graphically in Fig. 4.5. The semantics are formalized by the Q-Loop-Converge and Q-Loop-Unroll rules in Fig. 5.4.

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 [6], 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/syntactic loop unrolling — e.g. as applied 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. That is, the k-th demanded unrolling corresponds to the k-th application of the loop body's abstract semantics (i.e. the k-th abstract iteration) rather than the k-th concrete execution of the loop. As a result, it is sound with respect to the concrete semantics of the program under analysis and is guaranteed to converge, as we will show in the following section.

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:

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

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 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.

5.3.3 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. 5.5.

As described informally in Section 4.1.2, 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 and fourth premises ensure that DAIG well-formedness is preserved, by preventing emptying of source nodes and ill-typed edits respectively.

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., with $k > 1$) is edited since results from some previous iterations (up to that k) may not need to be discarded, but it simplifies the presentation and handles all program edits with maximal reuse.

5.4 Soundness, Termination, and From-Scratch Consistency

In this section, we state and prove three key properties of demanded abstract interpretation graphs:

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

E-Commit

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

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 5.5: 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 "Editing reference cell n of DAIG $\mathcal{D}$ with value v_ε yields updated DAIG $\mathcal{D}'$," where v_ε ranges over values v and the "empty" symbol ε.

- *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,
- *soundness*, which guarantees that query results over-approximate the concrete semantics, and
- *query termination*, which guarantees termination of the DAIG query semantics even in the presence of unbounded abstract loop unrolling.

All three results depend on the preservation of DAIG well-formedness (Definition 5.2), DAIG–CFG consistency (Definition 5.3), and DAIG–AI consistency (Definition 5.4) under queries and program edits.

Lemma 5.2 (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' content 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 $\left\{ \mathsf{incr\text{-}c}(c) \;\middle|\; \underline{\ell}^{(k-1)} \rightsquigarrow \mathtt{dest}(c) \rightsquigarrow \underline{\ell}^{(k)} \right\}$; 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 5.3 (DAIG–CFG Consistency Preservation)**.**
If $\mathcal{D} \approxeq \langle L, E, \ell_0 \rangle$ *then:*

- *if* $\mathcal{D}, M \vdash n \Rightarrow v \; ; \mathcal{D}', M'$ *then* $\mathcal{D}' \approxeq \langle L, E, \ell_0 \rangle$*;*

- *if* $\mathcal{D} \vdash n \Leftarrow s \; ; \mathcal{D}'$ *then* $\mathcal{D}' \approxeq \langle L, E', \ell_0 \rangle$*, where* E' *is* E *with the edit applied.*

The first bullet point states that DAIG–CFG consistency is preserved under DAIG evaluation, while the second states that a program edit (*not* an arbitrary edit to abstract state) in a DAIG consistent with an initial CFG yields a DAIG consistent with the corresponding edited CFG.

The post-edit CFG is defined as follows: since n names a reference cell of type *Stmt* in $\mathcal{D}$ and $\mathcal{D} \approxeq \langle L, E, \ell_0 \rangle$, n is either of the form $\underline{i} \cdot \underline{\ell} \cdot \underline{\ell'}$ or $\underline{\ell} \cdot \underline{\ell'}$, said reference cell holds some non-ε value $s' : \mathit{Stmt}$, and $\ell \dashv\!\lfloor\, s' \,\rfloor\!\mapsto \ell' \in E$. Thus, the post-edit CFG edges are $E' \triangleq E / \{\ell \dashv\!\lfloor\, s' \,\rfloor\!\mapsto \ell'\} \cup \{\ell \dashv\!\lfloor\, s \,\rfloor\!\mapsto \ell'\}$. That is, E' is E with whichever edge corresponds to DAIG name n re-labelled by s.

Note that this lemma as stated applies only to in-place edits of CFG edges, but the same technique applies for deletions (which can be seen as in-place edits where s is the no-op `skip` statement) and insertions (for which a new sub-DAIG is constructed and inserted, before dirtying proceeds as usual).

Proof (Bullet 1): 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 5.3 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. □

Proof (Bullet 2): The proof is straightforward: the only possible *structural* changes from the initial DAIG $\mathcal{D}$ to the post-edit DAIG $\mathcal{D}'$ are fix edges reset to $k = 1$ by rule E-Loop, which preserve condition (3) of Definition 5.3 for any back edges reachable from the edit. The only possible reference cell *content* changes from $\mathcal{D}$ to $\mathcal{D}'$ are dirtied abstract state cells and the edited statement cell $r = n[s : \mathit{Stmt}]$, whose value s agrees with the corresponding CFG edge in E' and thus satisfies the relevant $r \in R$ conjunct of whichever condition of Definition 5.3 applies to its edge type (i.e. straightline forward edge, join-point forward edge, or back edge). □

Lemma 5.4 (DAIG–AI Consistency Preservation)**.**
If $\mathcal{D} \approxeq \langle \Sigma^\sharp, \varphi_0, [\![\cdot]\!]^\sharp, \sqsubseteq, \sqcup, \nabla \rangle$ *and either* $\mathcal{D}, M \vdash n \Rightarrow v \; ; \mathcal{D}', M'$ *or* $\mathcal{D} \vdash n \Leftarrow s \; ; \mathcal{D}'$*, then* $\mathcal{D}' \approxeq \langle \Sigma^\sharp, \varphi_0, [\![\cdot]\!]^\sharp, \sqsubseteq, \sqcup, \nabla \rangle$.

Proof. The two cases in the premise of the lemma correspond to query evaluation and program edits.

For program edits, DAIG–AI consistency is clearly preserved:

- Reference cell $\underline{\ell_0}$ in $\mathcal{D}'$ is unchanged from $\mathcal{D}$ since it is not reachable from any statement-type reference cell (which n is guaranteed to be by the well-typedness premise of rule E-Commit).
- All in-degree ≥ 1 reference cells in $\mathcal{D}'$ are either empty (in which case they vacuously satisfy Def. 5.4) or non-empty (in which case they satisfy Def. 5.4 for $\mathcal{D}'$ because they did so for $\mathcal{D}$).

For query evaluation, we proceed 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 5.4's $f \neq \mathsf{fix}$ 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 5.4's $f = \mathsf{fix}$ case.

□

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 5.2 (DAIG From-Scratch Consistency)**.** *For all sound M and well-formed $\mathcal{D}$ such that $\mathcal{D} \approxeq \langle L, E, \ell_0 \rangle$ and $\mathcal{D} \approxeq \langle \Sigma^\sharp, \varphi_0, [\![\cdot]\!]^\sharp, \sqsubseteq, \sqcup, \nabla \rangle$, 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 $\mathcal{D} \approxeq \langle \Sigma^\sharp, \varphi_0, [\![\cdot]\!]^\sharp, \sqsubseteq, \sqcup, \nabla \rangle$, $v = [\![\ell]\!]^{\sharp *}$.

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 $\ell \in L_{\not\downarrow}$ 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 $\mathcal{D} \approxeq \langle \Sigma^\sharp, \varphi_0, [\![\cdot]\!]^\sharp, \sqsubseteq, \sqcup, \nabla \rangle$.

Case $\ell \in L_{\not\downarrow}$: Since $\mathcal{D} \approxeq \langle L, E, \ell_0 \rangle$, there exists an $\ell' \dashv [\, s \,] \mapsto \ell' \in E_f$, $\underline{\ell'} \cdot \underline{\ell}[s : \mathit{Stmt}] \in R$, 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 $\mathcal{D} \approxeq \langle L, E, \ell_0 \rangle$, there are k forward CFG edges into ℓ and for each such $\ell_i \dashv s_i \vdash \ell$, we have $\underline{i} \cdot \underline{\ell_i} \cdot \underline{\ell}[s_i : \mathit{Stmt}] \in R$ 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 $\mathcal{D} \approxeq \langle L, E, \ell_0 \rangle$, 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 $\underline{i} \cdot \underline{\ell_i} \cdot \underline{\ell}[s_i : \mathit{Stmt}] \in R$ 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 $\mathcal{D} \approxeq \langle L, E, \ell_0 \rangle$, ℓ is the destination of a CFG back-edge $\ell' \dashv s \vdash \ell \in E_b$ 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[7] 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 $\mathcal{D} \approxeq \langle L, E, \ell_0 \rangle$, 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 $\mathcal{D} \approxeq \langle \Sigma^\sharp, \varphi_0, [\![\cdot]\!]^\sharp, \sqsubseteq, \sqcup, \nabla \rangle$.

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

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

is therefore the fixed-point $[\![\ell]\!]^{\sharp *}$ — or Q-Loop-Unroll, in which case v is equal to $[\![\ell]\!]^{\sharp *}$ by the inductive hypothesis. □

Corollary 5.1. *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 (Theorem 5.1)) and a DAIG query for the abstract state at a location ℓ returns $[\![\ell]\!]^{\sharp *}$, DAIG query results themselves are sound.

Theorem 5.3 (DAIG Query Termination)**.** *For all M and well-formed $\mathcal{D}$ such that $\mathcal{D} \approxeq \langle L, E, \ell_0 \rangle$ and $\mathcal{D} \approxeq \langle \Sigma^\sharp, \varphi_0, [\![\cdot]\!]^\sharp, \sqsubseteq, \sqcup, \nabla \rangle$, 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 $\texttt{dest}(c) = n$. 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 $M_0 \triangleq M$.

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 $f = \mathsf{fix}$: 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. □

We have observed parallels between Theorems 5.2, 5.3, 5.4 and 5.3 and the classic progress and preservation lemmas of type safety: the first three theorems state the preservation of their respective properties, under which the fourth ensures that query progress is always possible.

This is in line with the design of DAIGs as reified states of abstract interpretation fixed-point computations, and of their semantics as a small-step operational specification of such a computation. By this analogy, these theorems are best understand as a proof that "analysis in well-formed consistent DAIGs doesn't go wrong".

5.5 Implementation and Evaluation

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

```
1  module type Dom = sig
2    type t                                  ⟨ Σ♯,
3    val init : t                              φ0,
4    val interpret : Stmt.t -> t -> t        [[·]]♯,
5    val implies : t -> t -> bool              ⊑,
6    val join : t -> t -> t                    ⊔,
7    val widen : t -> t -> t                   ∇ ⟩
8    val is_bot : t -> bool
9    include Adapton.Data.S with type t := t
10 end
```

Figure 5.6: Module argument signature for the DAIG functor. The `Adapton.Data.S` signature is provided by the `adapton.ocaml` library and defines `compare`, `equal`, and `hash` functions. Note the close correspondence between the operations and data types required of domain implementors and the abstract domain signature $\langle \Sigma^\sharp, \varphi_0, [\![\cdot]\!]^\sharp, \sqsubseteq, \sqcup, \nabla \rangle$.

- *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:* In these rich domains, what degree of performance improvement can be obtained by performing incremental and/or demand-driven analysis, as compared to batch analysis?

5.5.1 Implementation

The DAIG framework is implemented in approximately 2,500 lines of OCaml code (Stein et al., 2021c,b). 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. 5.4 semantics) is provided via `adapton.ocaml`, an open-source implementation of the technique of Hammer et al. (2015).

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 (shown in Fig. 5.6) 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 of this chapter is defined over intraprocedural control-flow graphs, the implementation evaluated in this section supports analysis of non-recursive programs with static calling semantics (i.e., no virtual dispatch or higher-order functions) by taking an operational approach based on context-sensitivity. (as described in Chapter 4)

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 (Sharir and Pnueli, 1981).

5.5.2 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 and show that they are capable of proving the same class of semantic properties as the underlying abstract interpreter. In **??**, 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 (Cousot and Cousot, 1977). An abstract state in the interval domain is a finite map from variables to intervals, with pointwise liftings of join, widen, and implication.

Abstract interpretation in this domain is known as interval analysis and is commonly used, e.g., to verify the safety of array accesses by showing that, at each array access `a[i]`, `i` takes on a value between 0 and the length of `a`. Interval analysis has been applied at industrial scale, for example by Cousot et al. (2005).

In practice, it is common to use an optimized off-the-shelf interval abstract domain such as that of APRON (Jeannet and Miné, 2009) or Elina (Singh et al., 2017). 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* – essentially by "wrapping" the APRON domain in our domain interface, requiring an effort comparable to doing so for existing batch frameworks.

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 (Santos, 2016).

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' (Reynolds, 2002),
- 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. (2007) (specialized to a fixed inductive definition for list segments). All told, the implementation of this shape domain requires approximately 500 lines of OCaml code.

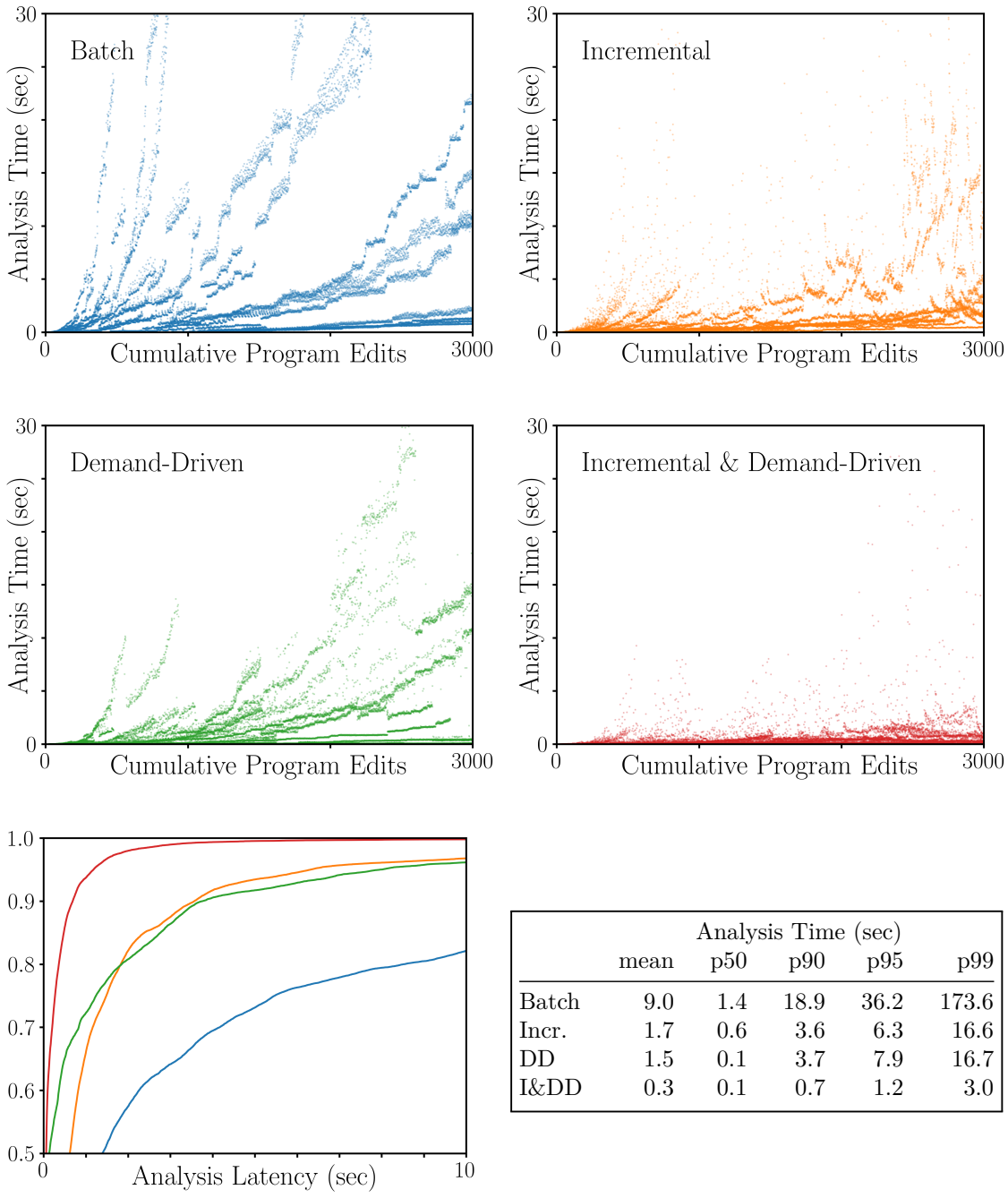
We have applied this DAIG-based shape analysis to successfully verify the correctness and memory-safety of the list `append` procedure of Fig. 4.1b, along with several linked list utilities from the aforementioned Buckets.js library including `foreach` and `indexof` (Santos, 2016). Analysis of the ℓ_3-to-ℓ_4-to-ℓ_3 loop of the list `append` procedure converges in one demanded unrolling with a precise result. This shape analysis instantiation provides additional evidence to the expressivity of the approach — that our framework supports standard implementations of such rich abstract domains.

5.5.3 Scalability

To study scalability, we conducted an initial investigation of what performance improvements are possible with demanded analysis variants in our framework. These potentials have not been previously studied, as previous incremental and/or demand-driven frameworks could not easily express this analysis.

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 (Miné, 2006). This domain has infinite height and therefore requires widening.

To exercise the analyses, we created synthetic workloads consisting of 3,000 random edits



	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 5.7: Performance of octagon analysis on the synthetic workload of interleaved program edits and analysis queries described in Section 5.5.3. 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.

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. 5.7, 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. 5.7 gives a cumulative distribution of analysis latencies, again showing the large advantage of the incremental demand-driven analysis over other configurations.

5.6 Conclusion

This chapter presents 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.

However, the techniques described here are not sufficiently expressive to tackle the complexities of real-world program analysis problems on their own. In particular, DAIGs are ill-equipped to handle interprocedural abstract interpretation in general. The formalism of Sections 5.2 and 5.3 talks exclusively about intraprocedural analysis problem over a single CFG, and the implementation and evaluation of Section 5.5 apply ad-hoc extensions to implement the operational approach over a limited class of procedures and calls. Nonetheless, DAIGs represent a promising advance over the state of the art in incremental and demand-driven analysis, and can serve as powerful building

blocks in more realistic interprocedural analysis infrastructure.

Chapter 6

Demanded Summarization and Interprocedural Dependency Tracking

This chapter considers the problem of *compositionality* in the context of demanded abstract interpretation. Compositionality is key for scalable interprocedural demanded analysis, as it enables reuse of already-computed analysis results for unmodified code. However, existing techniques for interactive static analysis either lack compositionality, cannot express arbitrary abstract domains, or do not provide formal guarantees on analysis behavior in the presence of recursion.

The previous chapter offers an approach for *intraprocedural* demanded analysis, making analyses with arbitrary abstract domains incremental and demand-driven by explicitly reifying the abstract interpretation computation into demanded abstract interpretation graphs (DAIGs).

A core obstacle in adapting DAIGs into a compositionally interprocedural framework is that DAIGs intuitively reify an "operational" abstract semantics for intraprocedural dataflow (i.e. how to *interpret* commands and compute the analysis state at a program point given analysis states at predecessor points), while procedure summaries represent a "denotational" abstract semantics, mapping code to a functional or relational abstraction of its effects (i.e. a *compilation* of code to a transformer on analysis states).

In this chapter, we bridge that operational–denotational gap with a technique that we call *demanded summarization*. At a high level, the operational DAIG-based analysis of a procedure is instantiated on demand to produce compositional, denotational summaries, drawing inspiration from tabulation-based approaches (Sharir and Pnueli, 1981; Reps et al., 1995). The key technical challenge is in the opposite direction: we also need to be able apply a denotational procedure summary in

the middle of an operational DAIG-based analysis with arbitrary abstract domains and recursive procedures — while getting the desired metatheoretical guarantees for sound and from-scratch consistent analysis results. To address this challenge, we define a *demanded summarization graph* (DSG) structure that is modified on-the-fly to capture the dynamic dependencies that arise from using and invalidating demanded procedure summaries.

These dynamic dependencies then enable our algorithm to detect self-referential procedure summaries resulting from analyzing recursive procedures that require a further fixed point iteration. Supporting the analysis of recursive procedures is particularly important in languages with higher-order or dynamic dispatch, as it is well-known that reasonable approximation in the underlying call graph construction can lead to recursion in the program abstraction.

Other recent compositional analysis frameworks (Blackshear et al., 2018; Calcagno and Distefano, 2011) incorporate some degree of incrementality, to reduce analysis times on CI servers after a change. Our framework supports a finer-grained incrementality and demand than these approaches (at the individual statement level), in support of interactive updates of results. Further, we provide a full metatheory for our framework showing soundness, termination, and from-scratch consistency in the presence of arbitrary abstract domains (including infinite-height domains with non-monotonic widening) and recursive procedures.

This chapter includes portions of a paper "Interactive Abstract Interpretation with Demanded Summarization" currently under submission with co-authors David Flores, Bor-Yuh Evan Chang and Manu Sridharan.

6.1 Concrete Syntax and Semantics

Recall from Fig. 5.1 that the DAIG-based analysis framework of Chapter 5 operated over a single program control-flow graph whose vertices are program locations and whose edges are labelled by statements.

In this section, we extend the concrete syntax and semantics of that control-flow graph language with procedural abstraction and procedure call primitives. We consider programs to be

labelled collections of procedures, defined as follows:

locations	$\ell \in \mathit{Loc}$	
statements	$s \in \mathit{Stmt}$	
procedure labels	$\rho \in \mathit{Label}$	$\ni \rho_{\text{main}}$
control-flow edges	$e \in \mathit{Edge}$	$::= \ell \dashv[\, s \,]\mapsto \ell'$
		$\mid \ell \dashv[\, \texttt{call}\ \rho \,]\mapsto \ell'$
~~programs~~ procedures	$\langle L, E, \sout{\ell_0} \rangle$	$: \mathit{Proc} = \mathcal{P}(\mathit{Loc}) \times \mathcal{P}(\mathit{Edge}) \sout{\times \mathit{Loc}}$
programs	P	$: \mathit{Label} \hookrightarrow \mathit{Proc}$

Figure 6.1: A program in this interprocedural imperative language is a labelled collection of procedures, each of which is a reducible flow graph whose vertices are program locations ℓ and whose edges are labelled either by program statements s or procedure calls `call` ρ.
Changes with respect to the language of Chapter 5 are highlighted: additions shaded in green and deletions struck through in red. Procedure-call edges have been added to the language, and programs are now labelled collections of CFGs rather than just one monolithic CFG.

Programs are finite maps from procedure labels ρ (including a distinguished "main" procedure ρ_{main}) to procedures. Each procedure $\langle L, E \rangle$ is a reducible control-flow graph over an unspecified statement language *Stmt* (as in Chapter 5) extended with procedure calls of the form `call` ρ. Statements are left unspecified as before to preserve genericity, with procedure calls treated separately to simplify the presentation.

We denote by L_P^ρ, E_P^ρ, $\mathsf{entry}_P(\rho)$ and $\mathsf{exit}_P(\rho)$ respectively the program-location set, control-flow edge set, entry location, and exit location of ρ. We also write ρ_P^ℓ for the unique ρ containing ℓ and E_P^* for the set of all control-flow graph (CFG) edges in P, and elide P subscripts where they are clear from context. This core language considers direct calls without parameters or return values for ease of presentation only but this is not a fundamental limitation; we discuss implementation considerations when applying our approach to real-world Java programs in **??**.

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

semantics for statements:

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

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

$$\begin{array}{lcl} [\![\, \mathsf{entry}_P(\rho_{\text{main}}) \,]\!]^*_P & \supseteq & \{(\sigma_0, \varepsilon)\} \\ [\![\, \ell \,]\!]^*_P & \supseteq & \left\{ ([\![s]\!]\sigma, \kappa) \;\middle|\; \begin{array}{l} \ell' \dashv [\, s \,] \mapsto \ell \in E^*_P \\ \quad \wedge\ (\sigma, \kappa) \in [\![\ell']\!]^*_P \end{array} \right\} \\ [\![\, \mathsf{entry}_P(\rho) \,]\!]^*_P & \supseteq & \left\{ (\sigma, e :: \kappa) \;\middle|\; \begin{array}{l} e = \ell \dashv [\, \texttt{call}\ \rho \,] \mapsto \ell' \in E^*_P \\ \quad \wedge\ (\sigma, \kappa) \in [\![\ell]\!]^*_P \end{array} \right\} \\ [\![\, \ell' \,]\!]^*_P & \supseteq & \left\{ (\sigma, \kappa) \;\middle|\; \begin{array}{r} (\sigma, \ell \dashv [\, \texttt{call}\ \rho \,] \mapsto \ell' :: \kappa) \\ \in [\![\mathsf{exit}_P(\rho)]\!]^*_P \end{array} \right\} \end{array}$$

Abstract Semantics. An abstract interpreter for *procedures* of this language is a tuple $\langle \Sigma^\sharp, \varphi_0, [\![\cdot]\!]^\sharp, \sqsubseteq, \sqcup, \nabla \rangle$ consisting of an abstract domain $\Sigma^\sharp$ equipped with distinguished initial state φ_0, partial order $\sqsubseteq$, join $\sqcup$, and widening ∇, as well as an abstract semantics (i.e., transfer function) $[\![\cdot]\!]^\sharp$ that interprets statements as functions over abstract states, all subject to the usual soundness conditions. The solution to an intraprocedural abstract interpretation problem is a fixed-point of the abstract transfer function over the flow graph $\langle L, E \rangle$, with the join $\sqcup$ applied at locations in L with in-degree ≥ 2 and the widening ∇ applied infinitely often on cycles. It is well-known that such a solution can be computed by worklist iteration, and that such a solution is sound if the transfer function is locally-sound (Cousot and Cousot, 1977).

Note that this intraprocedural abstract semantics is identical to that of Section 5.1: as

before, this interface is generic so as to preserve domain agnosticity and allow for arbitrary domains implemented in general purpose languages. The presentation is as standard as possible and is repeated here just to fix notation and terminology for clarity's sake.

Just as we defined the concrete state-collecting semantics $[\![\cdot]\!]^*_P$ for this language with procedure calls as a fixed-point over the concrete transfer function, we can now define an abstract state-collecting semantics $[\![\cdot]\!]^{\sharp *}_P$ as a fixed-point over abstract semantic functions. However, such a definition does not correspond to a computable analysis, since the space K of concrete stacks is unbounded. A standard solution to this issue is to apply the call-strings approach (Sharir and Pnueli, 1981), abstracting concrete stacks by truncating instead of extending them indiscriminately in the equation for procedure entry locations. Unfortunately, as argued in Chapter 4, the induced global dependency structure of context-sensitive analysis is not conducive to efficient incremental and demand-driven evaluation.

Demanded Abstract Interpretation. This section briefly summarizes some relevant details of the previous chapter. Recall that demanded abstract interpretation graph (DAIG) $\mathcal{D}$ reifies an intraprocedural abstract interpretation computation into a structure that supports interactive program analysis, that is, demand-driven queries and incremental edits (Stein et al., 2021a). Partial analysis results are stored in reference cells that are assigned unique names $n ::= \underline{\ell} \mid n_1 \cdot n_2 \mid \ldots$. The location name $\underline{\ell}$ corresponds to the cell that, when demanded, stores the fixed-point invariant at location $\underline{\ell}$, and the product name $\underline{\ell} \cdot \underline{\ell'}$ stores the statement s for the control-flow edge $\ell \dashv\! [\, s \,]\! \mapsto \ell'$; full definitions are given in the previous chapter.

A DAIG is then a directed acyclic hypergraph where reference cells are nodes and where edges are labeled by abstract interpreter operations (e.g., $[\![\cdot]\!]^\sharp$, $\sqcup$, ∇) that specify the operation to apply when its output cell is demanded. Demanded abstract interpretation is captured by the two DAIG-rewriting judgment forms presented in Chapter 5:

$$\text{demanding an invariant} \quad \mathcal{D} \vdash n \Rightarrow \varphi \; ; \mathcal{D}'$$

$$\text{dirtying on an edit} \quad \mathcal{D} \vdash n \Leftarrow v_\varepsilon \; ; \mathcal{D}'$$

The judgment form[1] $\mathcal{D} \vdash n \Rightarrow \varphi \; ; \mathcal{D}'$ is read as, "A query for the abstract state named by n in DAIG $\mathcal{D}$ yields result φ by demanded abstract interpretation with updated DAIG $\mathcal{D}'$." A demanded abstract interpretation computes a result φ to store in the cell named by n by evaluating backwards-reachable dependencies of cell n in DAIG $\mathcal{D}$ while unrolling fixed point computations on demand to maintain the DAIG acyclicity invariant and eventually resulting in $\mathcal{D}'$.

Queries are sound, terminating, from-scratch-consistent, and preserve the consistency of a DAIG with the structure of the underlying procedure and abstract semantics, all as shown in Section 5.4.

The judgment form $\mathcal{D} \vdash n \Leftarrow v_\varepsilon \; ; \mathcal{D}'$ is read as, "An edit that writes value v (or the empty symbol ε) to the cell named by n in DAIG $\mathcal{D}$ yields updated DAIG $\mathcal{D}'$ with cells depending on n marked as 'dirty'." Demanded abstract interpretation supports incremental edits by eagerly "dirtying" those results forward-reachable from the edit to cell n in DAIG $\mathcal{D}$ (e.g., editing statement $\not\S$ to s' in a cell $\underline{\ell \cdot \ell'}$ for the control-flow edge $\ell \dashv [\not\S \, s'] \mapsto \ell'$). Dirtying indicates those cells that need to be recomputed if later demanded. This dirtying judgment is conservative in the sense that all potentially-affected abstract states are dirtied.

Summarization and Tabulation. An interprocedural abstract interpretation that abstracts the concrete stack (e.g. with a call string) and *operationally* propagates abstract data flow in the resulting interprocedural flow graph can naturally be reified into a DAIG-like structure. Unfortunately, as illustrated in Chapter 4, the induced global dependency structure seems overly conservative for programs with good procedural abstraction. Instead, an alternative approach is to build compositional summaries for each procedure (i.e., the *denotational* or functional approach (Sharir and Pnueli, 1981)). Procedure summaries can be built by defining relational, two-state abstract domains (e.g., (Jeannet et al., 2010; Yorsh et al., 2008)) or by tabulating pairs of abstract states (Reps et al., 1995; Sharir and Pnueli, 1981). In either case, the goal is to compute procedure summaries $\{\varphi\} \; \rho \; \{\varphi'\}$ for each procedure ρ that can be applied at call sites $\ell \dashv [\texttt{call} \; \rho] \mapsto \ell'$.

[1] We elide the global memo table M from the evaluation semantics from this point on, as it is orthogonal to the ideas of demanded summarization presented in this chapter

A tabulation approach works by propagating intraprocedural dataflow through the procedure control-flow graph using the abstract transfer function $[\![\cdot]\!]^\sharp$, join $\sqcup$, and widen ∇ until it encounters a procedure call, at which point either (1) corresponding procedure summaries have already been computed and can simply be applied, or (2) new summaries are required, so the process continues recursively. Convergence is guaranteed through suitable applications of widening at recursive calls.

The algorithm operates over a worklist of $\Sigma^\sharp \times \mathit{Loc}$ pairs (i.e. a procedure-entry abstract state and a program location) and computes an invariant map I of type $\mathit{Loc} \times \Sigma^\sharp \mapsto \Sigma^\sharp$, wherein $I(\ell, \varphi)$ is an overapproximation of reachable states at ℓ from procedure-entry abstract state φ. An invariant $I(\ell, \varphi)$ is best understood as the postcondition of a Hoare triple with precondition φ over the valid paths to ℓ from the entry of its containing procedure. Thus, when ℓ is $\mathsf{exit}(\rho)$, the invariant $I(\ell, \varphi)$ is equivalent to a procedure summary $\{\varphi\}\ \rho\ \{I(\varphi, \ell)\}$, which can be used to interpret calls to ρ in compatible abstract states.

The algorithm begins with the worklist $\{(\varphi_0, \mathsf{entry}(\rho_{\text{main}})\}$ and the invariant map $\{(\varphi_0, \mathsf{entry}(\rho_{\text{main}})) \mapsto \varphi_0\}$ (i.e. the initial abstract state at program entry). Then, until the worklist is empty, it pops an element (φ, ℓ) and processes each $l \dashv[\, s \,]\vdash l' \in E^*$ as follows:

If s is not a procedure call, compute the local transfer function result $[\![s]\!]^\sharp I(\ell, \varphi)$, join/widen it as needed with existing abstract state $I(\ell', \varphi)$, and add (φ, ℓ') to the worklist if I was modified. Otherwise, when s is a procedure call, either:

- a compatible summary $I(\mathsf{exit}(\rho), \varphi'')$ (where $\varphi' \sqsubseteq \varphi''$) is available for each callee and callee-entry abstract state pair $(\rho, \varphi') \in \mathsf{call}^\sharp(s, I(\ell, \varphi))$, in which case they can be joined together and the result added to the invariant map and worklist.
- no such summary is available for at least one (ρ, φ'), in which case we add $(\varphi', \mathsf{entry}(\rho))$ to the worklist, set $I(\mathsf{entry}(\rho), \varphi')$ to φ', and return to this callsite once those callee summaries are computed. If the call is recursive — i.e. ρ and ρ^ℓ belong to a callgraph cycle — then using $\varphi \nabla \varphi'$ instead of φ' is sufficient to ensure convergence.

6.2 Demanded Summarization

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

So instead of constructing a single DAIG corresponding to an interprocedural control-flow graph that results from connecting call sites $\ell \dashv [\, \texttt{call}\ \rho \,] \mapsto \ell'$ to procedures ρ (i.e., with ℓ transitioning to $\mathsf{entry}(\rho)$ and $\mathsf{exit}(\rho)$ to ℓ') as alluded to in Section 6.1, we consider DAIGs for each procedure that are instantiated on demand.

In particular, we can instantiate an initial DAIG $\mathcal{D}^{\mathsf{init}}_{\rho,\varphi}$ for any procedure ρ with initial abstract state φ stored in cell $\underline{\mathsf{entry}(\rho)}$ and with otherwise uninitialized abstract-state cells using the $\mathcal{D}_{\mathsf{init}}$ construction of Definition 5.5, i.e.

$$\mathcal{D}^{\mathsf{init}}_{\rho,\varphi} \triangleq \mathcal{D}_{\mathsf{init}}\left(\langle L^\rho, E^\rho, \mathsf{entry}(\rho)\rangle \ ,\ \langle \Sigma^\sharp, \varphi, [\![\cdot]\!]^\sharp, \sqsubseteq, \sqcup, \nabla\rangle\right)$$

However, the $\mathcal{D}_{\mathsf{init}}$ construction of Definition 5.5 has no mechanism to reify procedure-call CFG edges of the form $\ell \dashv [\, \texttt{call}\ \rho \,] \mapsto \ell'$, which may occur in E^ρ. To handle them, we extend slightly the syntax and semantics of DAIGs as follows. In Fig. 6.2, we show how a CFG edge $\ell \dashv [\, \texttt{call}\ \rho \,] \mapsto \ell'$ is encoded by a new kind of DAIG edge with label ρ connecting the abstract state reference cell $\underline{\ell}$ to that of $\underline{\ell'}$. This label ρ states that the function to compute $\underline{\ell'}$ from $\underline{\ell}$ is described by a summary of procedure ρ, which we can view as a (higher-order) reference to another DAIG for ρ.

Since DAIG semantics are concerned only with *intra*procedural abstract interpretation, they also have no means to evaluate such an edge. As such, queries can now get "stuck" with a value for $\underline{\ell}$ but no way to compute a value for $\underline{\ell'}$. To capture this possibility, we introduce a judgment form

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

which indicates that a query for the value named by n in $\mathcal{D}$ is stuck, unable to compute (and store at n') the result of a call to ρ with abstract state φ and where $\mathcal{D}'$ reflects any intraprocedural analysis

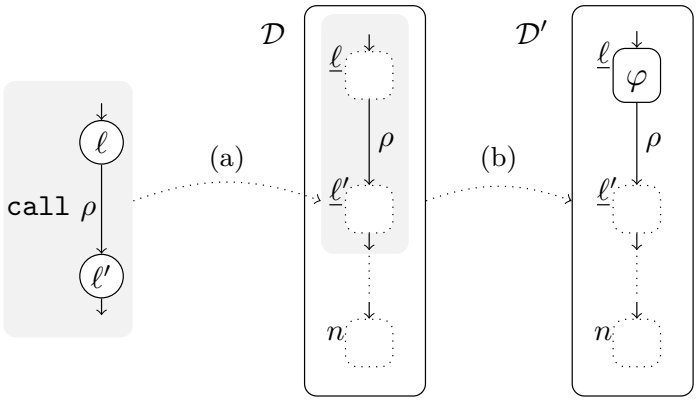


Figure 6.2: Translating a procedure-call CFG edge into a higher-order DAIG edge. Part (a) shows how a CFG edge of the form $\ell \dashv [\,\texttt{call}\ \rho\,] \mapsto \ell'$ is encoded into a corresponding DAIG edge labeled by ρ (both in shaded boxes). Intuitively, a procedure-labeled DAIG edge corresponds to computing a summary of procedure ρ by instantiating a DAIG for ρ as needed. Part (b) shows the effect of a subsequent query for a cell named n that depends transitively on the value at ℓ'. The query for the value of n is blocked by needing to apply a summary for ρ, which is captured by the judgment shown above demanding a summary.

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

Figure 6.3: A demanded summarization graph (DSG) $\mathcal{G}$ is a collection $\mathcal{D}^*$ of intraprocedural DAIGs overlaid by an interprocedural summary dependency map Δ. The summary dependency map Δ captures the essence of demanded summarization; it is dynamically extended to capture the dependencies from using demanded procedure summaries during demand-driven query evaluation that are later needed for incremental dirtying.

evaluation in $\mathcal{D}$ before it got stuck on the ρ-labelled edge. So evaluating a DAIG $\mathcal{D}$ for a cell n results in either an invariant to store in n or getting stuck in demanding a summary.

In the rest of this section, we introduce *demanded summarization graphs* (DSGs), which govern demand for callee summaries and thus enable intraprocedural abstract interpretation to get "unstuck" at call sites. In particular, evaluating DSGs interleaves intraprocedural dataflow analysis via DAIG evaluation with interprocedural tabulation via DAIG instantiation.

A demanded summarization graph (DSG) $\mathcal{G} = \langle \mathcal{D}^*, \Delta \rangle$ consists of a DAIG $\mathcal{D} = \mathcal{D}^*(\rho, \varphi)$ for each partially- or fully-computed summary of a procedure ρ with initial abstract state φ. Note that an instantiated procedure DAIG $\mathcal{D}$ is indexed by the pair of the procedure label ρ *and* an initial abstract state φ (or pre-condition). Thus, there may be multiple instantiated DAIGs for a procedure ρ with different initial abstract states. Crucially for incremental analysis (cf. Section 6.2.2), it also has a summary dependency map Δ that records interprocedural analysis dependencies from applying procedure summaries. A summary dependency δ has the form $(\rho', \varphi') \overset{n}{\hookleftarrow} (\rho, \varphi)$, which indicates that the return-site abstract state named by n in $\mathcal{D}^*(\rho', \varphi')$ depends upon the summary DAIG $\mathcal{D}^*(\rho, \varphi)$.

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

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

In Section 6.2.1, we define demand-driven query evaluation using the judgment form

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

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

Query evaluation is closely related to procedure summarization so we also define here the $\mathcal{G} \vdash \{\varphi\}\ \rho\ \{\varphi'\} \,;\, \mathcal{G}'$ judgment form, which makes explicit our interpretation of DAIG analysis results as composable procedure summaries.

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

$$\text{S-Query}\quad \frac{\mathcal{G} \vdash_{\varphi} \mathsf{exit}(\rho) \Downarrow \varphi' \,;\, \mathcal{G}'}{\mathcal{G} \vdash \{\varphi\}\ \rho\ \{\varphi'\} \,;\, \mathcal{G}'}$$

In particular, the S-Query rule shows how a summary $\{\varphi\}\ \rho\ \{\varphi'\}$ (i.e., a Hoare triple) for the procedure ρ is implied by an analysis result computed at the exit of the corresponding DAIG (i.e., $\mathcal{G} \vdash_{\varphi} \mathsf{exit}(\rho) \Downarrow \varphi' \,;\, \mathcal{G}'$). Intuitively, this rule captures reifying the denotational procedure summary through the operational fixed-point computation therein, applying local transfer functions and/or summary transformers.

6.2.1 Demand-Driven Query Evaluation in DSGs

Abstract interpretation in DSGs is *demand-driven* by default: analysis results are computed only as needed to answer queries. A *query* requests the abstract state at a specific program location ℓ under some procedure-entry precondition φ, and may be issued directly by a developer through their IDE, programmatically by a client analysis, or internally in service of another query.

Queries for analysis results in DSGs are governed by the $\mathcal{G} \vdash_{\varphi} \ell \Downarrow \varphi' \,;\, \mathcal{G}'$ judgment form, defined inductively by the Q-* inference rules of Fig. 6.4.

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

Q-Instantiate

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

Q-Delegate

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

Q-Apply-Summary

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

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

SQ-Other-Proc

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

SQ-Other-Pre

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

SQ-Self

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

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

F-Converge

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

F-Step

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

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

Figure 6.4: Operational semantics rules governing *queries* in DSGs. The judgment form $\mathcal{G} \vdash_\varphi \ell \Downarrow \varphi' \,;\, \mathcal{G}'$ is read as, "A query against DSG $\mathcal{G}$ for the abstract state at location ℓ with procedure-entry abstract state φ yields result φ' and updated DSG $\mathcal{G}'$." It is defined with a judgment form $\mathcal{G} \vdash (\rho, \varphi) \overset{?n}{\Longrightarrow} (\rho', \varphi') \,;\, \mathcal{G}'$ for demanding and applying summaries, and a judgment form $\mathcal{G} \vdash \mathsf{fix}(\rho, \varphi); \mathcal{G}'$ for fixed-point computations on self-referential summaries of recursive procedures.
The shorthand $\mathsf{R}_{\rho,\varphi}(\mathcal{G})$ defines the set of recursive-call return sites in the DAIG for procedure ρ with pre-condition φ that do not over-approximate the procedure-exit abstract state.

This judgment is defined using the system of three boxed judgment forms shown in Fig. 6.4, whose mutual induction implements the arbitrary interleaving between operational intraprocedural analysis, denotational compositional analysis, and fixed-point computation in recursive procedures. The three boxed judgments respectively handle *queries* $\mathcal{G} \vdash_{\varphi} \ell \Downarrow \varphi' ; \mathcal{G}'$ for abstract state (implemented by the Q-* inference rules), *summary queries* $\mathcal{G} \vdash \{\varphi\}\, \rho\, \{\varphi'\} ; \mathcal{G}'$ to handle procedure call semantics (implemented by the SQ-* inference rules), and *fixed-point queries* $\mathcal{G} \vdash \mathsf{fix}(\rho, \varphi); \mathcal{G}'$ to analyze self-referential summaries (implemented by the F-* inference rules).

Abstract State Queries. At a high level, procedure DAIGs are instantiated on demand with Q-Instantiate, queries are delegated to the appropriate DAIG with Q-Delegate, and whenever query evaluation in the underlying DAIG gets "stuck" the Q-Apply-Summary rule steps in to demand a procedure summary, apply it, and continue query evaluation in the DAIG.

More specifically, if the current DSG does not have a procedure DAIG for ρ^ℓ with pre-condition φ (i.e., $(\rho^\ell, \varphi) \notin \mathsf{dom}(\mathcal{D}^*)$), Q-Instantiate instantiates a new DAIG $\mathcal{D}^{\mathsf{init}}_{\rho^\ell, \varphi}$ for ρ^ℓ initializing the input abstract state to φ before reissuing the query for location ℓ in this extended DSG.

The Q-Delegate rule applies when the relevant DAIG is already available and can handle the query on its own. Note that, if the queried abstract state had previously been computed, no analysis computation is performed and the analysis state is unchanged.

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

$$\text{resolving summaries } \; \mathcal{G} \vdash (\rho, \varphi) \overset{?n}{\Longrightarrow} (\rho', \varphi') ; \mathcal{G}' \; .$$

which is read "in DSG $\mathcal{G}$ and in procedure ρ with pre-condition φ, return site n is resolved with a demanded summary for ρ' with pre-condition φ', yielding updated DSG $\mathcal{G}'$." Here in the Q-ApplySummary rule, it demands the summary for ρ with a pre-condition compatible with φ'

at the call site to resolve return site n in procedure-DAIG indexed by (ρ^ℓ, φ). Once a compatible summary is applied for the call to ρ in the updated DSG $\mathcal{G}'$, the query for location ℓ is reissued, just like with Q-Instantiate and Q-Delegate.

Summary Queries. The complexity lies in demanding summaries with the judgment form $\mathcal{G} \vdash (\rho, \varphi) \overset{?n}{\Longrightarrow} (\rho', \varphi') \,;\, \mathcal{G}'$, defined inductively with the SQ-* rules that we discuss here. Let us first consider the simplest case: a non-recursive[2] call to a procedure ρ' different from the current caller ρ, which we express with $\rho \neq \rho'$ in SQ-Other-Proc. In this case, in the second premise, we demand a Hoare-style summary $\{\varphi'\}\ \rho'\ \{\varphi_{\mathsf{post}}\}$ of the callee ρ' with the given abstract precondition φ'. Then, we simply write φ_{post} to the return site n in the third premise and add the fine-grained, demanded summarization dependency $(\rho, \varphi) \overset{n}{\hookleftarrow} (\rho', \varphi')$ to $\mathcal{G}''$, indicating that the value at the return site n now depends on the $\{\varphi'\}\ \rho'\ \{\varphi_{\mathsf{post}}\}$ summary. This dependency edge ensures that if this summary is somehow invalidated with incremental edits, then the return site n will also be invalidated.

The next two rules SQ-Other-Pre and SQ-Self implement a demanded tabulation with (directly) recursive procedures. In both rules, we are deriving judgments for when a query in (ρ, φ) depends on a summary of the same procedure ρ.

The SQ-Other-Pre rule applies when the call-site state φ' is not included in the pre-condition φ of this procedure DAIG (i.e., $\varphi' \not\sqsubseteq \varphi$). So, we demand another summary of ρ with widened pre-condition $\varphi \nabla \varphi'$, which yields post-condition φ_{post} and allows analysis to proceed in the same manner as for non-recursive calls.

This demanded summary (with precondition $\varphi \nabla \varphi'$) is guaranteed to be compatible with the callsitte being analyzed (where the abstract state is φ'), and widening ensures that only finitely many summaries are demanded for the recursive procedure ρ. Intuitively, applying SQ-Other-Pre corresponds to a demanded unrolling of recursive calls a finite (but *a priori* unbounded) number of times determined by the widening operator ∇.

If the call-site state φ' *is* included in the pre-condition φ of this procedure DAIG, the summary

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

of procedure ρ that we need is the one that we are currently demanding and SQ-Self applies. When a procedure summary is *self-referential* in this sense, some special care must be taken to compute a fixed point along the control-flow cycle(s) formed between the procedure exit and recursive return site(s). This fixed point computation is implemented by a judgment form for

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

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

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

Now, this fixed-point computation in a self-referential procedure summary is relatively straightforward. At each step, we demand a summary for procedure ρ with pre-condition φ, that is, $\mathcal{G} \vdash \{\varphi\}\, \rho\, \{\varphi'\}; \mathcal{G}'$. The shorthand $\mathsf{R}_{\rho,\varphi}(\mathcal{G}')$ then yields the set of recursive return sites whose abstract state is not over-approximated by the procedure-exit state φ' in the procedure DAIG indexed by (ρ, φ). When this set is empty (F-Converge), this fixed-point iteration has converged. Otherwise, some return sites n_i have abstract state φ_i not over-approximated by the procedure-exit state φ', so F-Step applies, widening φ' onto each return-site state before taking another step in the fixed-point iteration. Importantly, the widened state $\varphi_i \nabla \varphi'$ is written to cell n using the dirtying-on-an-edit judgment, meaning forward-reachable nodes from n in the procedure DAIG get dirtied.

These steps can be seen applied to the overview example in Fig. 4.9, where (a) applies SQ-Self, (b) and (c) apply F-Step, and F-Converge applies in the rightmost (final) state.

Note that procedure summarization, abstract state queries, summary queries, and self-referential summary fixed-points (the four boxed judgments of this section) are mutually recursively defined, so any number of summaries can be produced, cached and/or memo-matched upon in the

process of evaluating queries and tabulating summaries.

6.2.2 Incremental Edits in DSGs

When the program under analysis is edited, we must discard those analysis results that are potentially affected while retaining as many cached results as possible for future incremental reuse. This operation is given by an operational semantics over analysis states $\mathcal{G}$, just as with demand-driven query evaluation. The judgment form $\mathcal{G} \vdash_\rho n \Leftarrow s \,;\, \mathcal{G}'$ of Fig. 6.5 defines the impact of an edit that modifies the statement named by n in procedure ρ, discarding facts from DSG $\mathcal{G}$ to yield $\mathcal{G}'$. Note that although this judgment as-written only applies to CFG-structure-preserving statement *modifications*, the extension to statement *insertions* and *deletions* is straightforward: we insert or remove the DAIG region corresponding to the edit, then dirty $\mathcal{G}$ from its exit (merging the entry and exit location of deleted regions).

The program edit rule E-Delegate deals with the fact that there may be multiple (partially or fully computed) summary DAIGs for the edited procedure ρ, whereas the dirtying rules D-* perform the actual traversal of demanded summarization dependencies. Therefore, E-Delegate is simple: in order to apply an edit in ρ, we use the D-* rules to delegate that edit to each currently-instantiated DAIG over ρ.

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

Bridging the gap between operational demand-driven evaluation and applying denotational transformer summaries exposes the question of what is "minimal demand." For example, we now have a choice in when to dirty across demanded summarization dependencies in that the caller depends on the demanded summary of the callee, not the underlying procedure implementation. If

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

E-DELEGATE

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

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

D-DEMANDEDSUMMARIES

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

D-DAIG

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

Figure 6.5: Operational semantics rules governing *edits* to a program under analysis. The *program-edit* judgment form $\mathcal{G} \vdash_{\rho} n \Leftarrow s \,;\, \mathcal{G}'$ is read as "$\mathcal{G}$ is updated to $\mathcal{G}'$ by an edit that writes statement s at the position named by n in procedure ρ", and is defined in terms of the *dirtying* judgment form $\mathcal{G} \vdash_{\rho,\varphi} n \Leftarrow s_\varepsilon \,;\, \mathcal{G}'$, which applies edits or propagates changes across demanded summarization dependencies.

the callee is edited, what we have described here is dirtying to the caller, as the summary on which it depends has been invalidated. However, it is also possible that by re-demanding the exit location of the callee, we might discover that this particular edit does not change the post-condition of the summary on which the caller depends and dirtying does not need to continue back to the caller. At the same time, this re-demanding may not have been needed depending on the unknown future queries.

6.3 Demanded Summarization Metatheory

We now define and prove several critical meta-theoretic properties of demanded summarization graphs, namely *termination* of queries and edits, *from-scratch consistency* with the underlying batch abstract interpreter, and *soundness* with respect to the concrete semantics. Together, these results provide a strong theoretical foundation for demanded interprocedural abstract interpretation with DSGs, showing that analysis will not diverge and that no precision is lost in the lifting from batch to demanded analysis.

6.3.1 Consistency

First, we establish consistency lemmas analogous to those shown for DAIGs in Chapter 5. A DSG $\mathcal{G}$ is consistent with some underlying program P when each of its

Definition 6.1 (DSG Semantic Consistency)**.** *We say that a DSG $\mathcal{G} = \langle \mathcal{D}^*, \Delta \rangle$ is* semantically consistent *when it is syntactically well-formed and consistent with the program structure P and underlying abstract interpretation semantics $\langle \Sigma^\sharp, \varphi_0, [\![\cdot]\!]^\sharp, \sqsubseteq, \sqcup, \nabla \rangle$.*

(1) Each constituent DAIG $\mathcal{D}^*(\rho, \varphi)$ is well-formed (Definition 5.2) and consistent with the corresponding procedure CFG $\langle L_P^\rho, E_P^\rho, \mathsf{entry}_P(\rho) \rangle$ (Definition 5.3)and intraprocedural abstract semantics[3] $\langle \Sigma^\sharp, \varphi, [\![\cdot]\!]^\sharp, \sqsubseteq, \sqcup, \nabla \rangle$ (Definition 5.4).

[3] Note that the procedure-entry abstract state φ replaces the global program-entry abstract state φ_0 in the abstract interpreter interface here

(2) Each return-site abstract state has a corresponding dependency edge from its callee in Δ. That is, if n names a non-empty ref cell in some $\mathcal{D} = \mathcal{D}^*(\rho, \varphi)$ with a ρ'-labelled edge to n from n', then either

- $(\rho, \varphi) \overset{n}{\hookleftarrow} (\rho', \mathcal{D}(n')) \in \Delta$ *(when $\rho \neq \rho'$) or*
- $(\rho, \varphi) \overset{n}{\hookleftarrow} (\rho', \varphi \nabla \mathcal{D}(n')) \in \Delta$ *(when $\rho = \rho'$).*

(3) Dependency edges $(\rho', \varphi') \overset{n}{\hookleftarrow} (\rho, \varphi)$ in Δ are consistent with the relevant DAIGs, in that

- $\mathcal{D}^*(\rho', \varphi')(n) = \mathcal{D}^*(\rho, \varphi)(\mathsf{exit}(\rho))$ *(when $\rho \neq \rho'$) or*
- $\mathcal{D}^*(\rho', \varphi')(n) \sqsupseteq \mathcal{D}^*(\rho, \varphi)(\mathsf{exit}(\rho))$ *(when $\rho = \rho'$).*

(4) Analysis results in DAIGs are equal to the corresponding invariants produced by batch tabulation, where the domains coincide: $\mathcal{D}^{\mathcal{G}}_{\rho,\varphi}(\underline{\ell}) = I(\varphi, \ell)$ for all $\mathcal{D}^*(\rho, \varphi)(\underline{\ell})$ where $(\varphi, \ell) \in \mathsf{dom}(I)$.

Lemma 6.1 (Initial DSG Consistency)**.** *The DSG $\langle \varepsilon, \varepsilon \rangle$ with no cached results is semantically consistent.*

Proof. All four conditions of Definition 6.1 are vacuously satisfied, as they are universally quantified over $\mathcal{D}$ or Δ which are empty here. □

Lemma 6.2 (Consistency Preservation)**.** *If $\mathcal{G}$ is semantically consistent (with respect to a program P) then:*
if $\mathcal{G} \vdash_{\varphi} \ell \Downarrow \varphi' ; \mathcal{G}'$ then $\mathcal{G}'$ is semantically consistent, and
if $\mathcal{G} \vdash_{\rho} n \Leftarrow s ; \mathcal{G}'$ then $\mathcal{G}'$ is semantically consistent (with respect to the edited version of P).

Proof. We will show that each of the the four conditions of Definition 6.1 is preserved in turn.

(1) *Each $\mathcal{D} \in \mathcal{D}^*$ is well-formed and consistent.* This follows directly from Lemmas 5.2 to 5.4, since the only way we instantiate and modify DAIGs is through the inference rules defined there (modulo ρ-labelled edges, over which DAIG well-formedness makes no claims, but we will handle in the third bullet point).

(2) *Each return-site abstract state has in Δ a corresponding dependency edge from its callee.* The only query rules that modify return site abstract states either add the requisite edge to Δ (SQ-Other-Proc, SQ-Other-Pre, SQ-Self) or have a premise that guarantees the requisite edge is in Δ (F-Step, via $\mathsf{R}_{\rho,\varphi}$).

The dirtying rule D-DemandedSummaries empties a return-site n' and drops the corresponding edge from Δ, while D-DAIG throws away any dangling dependency edges Δ' after dirtying intraprocedurally.

(3) *Each edge in Δ is consistent with the values on either side.* First, note that by the same argument as the previous case, dependency edges are always removed when the return site they point to is dirtied. Then, we need only consider the SQ-* rules which add dependency edges to Δ: SQ-Other-Proc, SQ-Other-Pre, and SQ-Self. The dependency edges added in SQ-Other-* both satisfy the condition, since the value φ_{post} at the callee exit is written directly to the return site n.

The edge added in SQ-Self temporarily violates the condition, but the conclusion of its final premise guarantees that the resulting $\mathcal{G}'$ satisfies it (by the $\mathsf{R}_{\rho,\varphi}(\mathcal{G}')$ premise of F-Converge).

(4) *Cached results agree with batch tabulation.*

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

- Case D-DAIG: Since there are no dependency edges on this procedure in Δ, the only cached analysis results affected by the edit are in the DAIG $\mathcal{D}^*(\rho, \varphi)$. By (Stein et al., 2021a), the local dirtying in that DAIG is sound.
- Case D-DemandedSummaries: Because the dependency edge $(\rho', \varphi') \overset{n'}{\hookleftarrow} (\rho, \varphi)$ is in Δ, the analysis result at n' in $\mathcal{D}^*(\rho', \varphi')$ relied on this summary. By the inductive hypothesis and

preservation of the second condition of semantic consistency, the first premise dirties all results that depended transitively on that result, producing a $\mathcal{G}$ whose dependency map reflects all other uses of this summary. By the inductive hypothesis, the second premise also preserves cache agreement with batch tabulation, so the final resulting $\mathcal{G}'$ contains only those results that did not depend on the edit.

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

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

□

6.3.2 Termination

Theorem 6.1 (Termination)**.** *Queries and edits terminate:*

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

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

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

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

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

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

- *Base case* ($tc(\rho^\ell) = 0$, non-recursive ρ^ℓ): There are no calls in ρ^ℓ, so Q-Delegate applies, its premise guaranteed by intraprocedural DAIG query termination (Stein et al., 2021a).

- *Base case* ($tc(\rho^\ell) = 0$, recursive ρ^ℓ): A query against the relevant sub-DAIG either returns a result (i.e. $\mathcal{D}^{\mathcal{G}}_{\rho^\ell,\varphi} \vdash \underline{\ell} \Rightarrow \varphi' \,;\mathcal{D}'$) or is blocked at a recursive callsite (i.e. $\mathcal{D}^{\mathcal{G}}_{\rho^\ell,\varphi} \vdash \underline{\ell} \overset{?n}{\Longrightarrow} (\rho, \varphi) \,;\mathcal{D}'$).

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

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

  (2) $\mathcal{D}^{\mathcal{G}}_{\rho^\ell,\varphi} \vdash \underline{\ell} \overset{?n}{\Longrightarrow} (\rho^\ell, \varphi') \,;\mathcal{D}'$:

  Only Q-Apply-Summary applies, so we must derive a summary query.

  Since $\rho^\ell = \rho^\ell$, we consider only SQ-Other-Pre and SQ-Self, depending on whether or not $\varphi' \sqsubseteq \varphi$.

  - SQ-Self: We proceed through the F-* rules for fixed point computation to derive the final premise. Due to the convergence condition of ∇, F-Step can only apply finitely many times before F-Converge applies. The $\mathcal{G} \vdash \{\varphi\}\,\rho\,\{\varphi'\}\,;\mathcal{G}'$ premises of the F-* rules similarly converge either via Q-Delegate, or finitely many applications of Q-Apply-Summary (since there are finitely many syntactic callsites in a given procedure).

- SQ-Other-Pre: By the convergence condition of ∇, only finitely many new DAIGs over ρ^ℓ may be instantiated via the $\mathcal{G} \vdash \{\varphi \nabla \varphi'\}\ \rho\ \{\varphi_{\mathsf{post}}\}\ ;\mathcal{G}'$ premise, eventually yielding one where SQ-Self applies instead of SQ-Other-Pre and terminates by the argument of the previous case, allowing any intermediate ρ DAIGs also to terminate via intraprocedural analysis with Q-Delegate after their demanded summaries return.

- *Inductive case*($tc(\rho^\ell) = n$, non-recursive ρ^ℓ): A query for $\underline{\ell}$ under precondition φ against the relevant sub-DAIG $\mathcal{D}^{\mathcal{G}}_{\rho^\ell,\varphi}$ either returns a result (in which case Q-Delegate applies and the query terminates), or is blocked at some callsite (i.e. $\mathcal{D}^{\mathcal{G}}_{\rho^\ell,\varphi} \vdash \underline{\ell} \overset{?n}{\Longrightarrow} (\rho, \varphi')\ ;\mathcal{D}'$), in which case the only applicable rule is Q-Apply-Summary.

  We can derive the summary premise $\mathcal{G} \vdash \{\varphi_{\mathsf{pre}}\}\ \rho\ \{\varphi_{\mathsf{post}}\}\ ;\mathcal{G}'$ of Q-Apply-Summary via S-Query, using the inductive hypothesis to derive its premise because $tc(\rho) < n$. After writing the resulting φ_{post} to n in $\mathcal{D}'$ and updating Δ accordingly, we reissue the query for ℓ under φ (i.e. the recursive final premise of Q-Apply-Summary)

  Note that the inductive hypothesis does not apply here, since the reissued query is in the same procedure ρ^ℓ. However, since ρ^ℓ is nonrecursive, either Q-Delegate or Q-Apply-Summary must apply again.

  If it is Q-Delegate, then the query terminates with its result. On the other hand, if it is Q-Apply-Summary, then that reissued query may itself reissue a query for which Q-Apply-Summary applies, and so on. Note, though, that each time the query is reissued it is against a DSG with one fewer empty return site abstract state reference cell in its (ρ^ℓ, φ) DAIG. Thus, since there are only finitely many return sites in ρ^ℓ, eventually Q-Delegate will apply and the query will terminate.

- *Inductive case*($tc(\rho^\ell) = n$, recursive ρ^ℓ):

  This case is shown by combining the arguments of the previous two cases: the inductive hypothesis ensures termination of any sub-queries at non-recursive calls (as in the non-recursive inductive

case), while widening ensures termination of any sub-queries at recursive calls and convergence of the in-place fixed point computation performed by F-Step and F-Converge (as in the recursive base case).

□

And next, for edits:

Proof. Because $\mathcal{D}^*$ and therefore k are finite, this amounts to showing that each of the k-indexed dirtying premises terminates, i.e. that for all ρ, φ, n, s, and semantically consistent $\mathcal{G}$, there exists $\mathcal{G}'$ such that $\mathcal{G} \vdash_{\rho,\varphi} n \Leftarrow s \,;\, \mathcal{G}'$. We proceed by induction on the size of the dependency map Δ. In the base case when $\Delta_{\rho,\varphi}$ is empty (as is necessarily the case when $|\Delta| = 0$), D-DAIG applies — we dirty the $\mathcal{D}^*(\rho, \varphi)$ from n to produce $\mathcal{D}'$ (which terminates by Theorem 5.3), discard any dependencies Δ' of dirtied local results, construct $\mathcal{G}' = \langle \mathcal{D}^*[\mathcal{D}'/(\rho, \varphi)], \Delta/\Delta' \rangle$, and terminate.

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

6.3.3 From-Scratch Consistency and Soundness

Theorem 6.2 (From-Scratch Consistency)**.** *Query results are equal to the corresponding invariant computed by tabulation: if $\mathcal{G}$ is semantically consistent, $\mathcal{G} \vdash_{\varphi} \ell \Downarrow \varphi' \,;\, \mathcal{G}'$, and $(\varphi, \ell) \in \mathsf{dom}(I)$ then $\varphi' = I(\varphi, \ell)$.*

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

(1) the fourth property of semantic consistency ensures that DAIG results are consistent with tabulation results,

(2) Q-Delegate (which reads its result directly from the relevant procedure DAIG) is the only $\mathcal{G} \vdash_{\varphi} \ell \Downarrow \varphi' \,;\, \mathcal{G}'$ rule with no inductive premise, and

(3) the other two rules $\mathcal{G} \vdash_{\varphi} \ell \Downarrow \varphi' ; \mathcal{G}'$ return the result of their inductive premise,

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

Corollary 6.1 (Soundness)**.** *Query results are sound with respect to the concrete semantics: If* $\mathcal{G} \vdash \ell \Downarrow \varphi ; \mathcal{G}'$ *and* $\mathcal{G}$ *is semantically consistent then* $\sigma \models \varphi$ *for all* $\sigma \in [\![\ell]\!]^*$

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

6.4 Implementation and Evaluation

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

Although previous work on incremental and demand-driven program analysis has demonstrated considerable performance benefits, it has typically been applied to carefully restricted programming languages or synthetically-generated program edits (Stein et al., 2021a; Szabó et al., 2021). Our aim is to show that demanded summarization can provide comparable benefits, while also supporting more realistic programs and the complexities that come with them.

6.4.1 Implementation

To this end, we implemented a prototype analysis framework in approximately 7000 lines of OCaml code, split roughly evenly between frontend infrastructure and analysis logic (Stein et al., 2021c). The framework's frontend, intermediate representations, and core analysis engine are all designed to support incremental edits and demand queries.

Frontend. We use the `tree-sitter` incremental parsing library to interpret source-level changes at the granularity of concrete syntax tree nodes (Brunsfeld et al., 2021). In practice, we consider program edits that add or delete procedures or modify their headers; add, modify, or delete statements; and modify the headers of loops and conditionals.

Table 6.1: Statistics about programs, edits, and analysis thereof, showing the relative degrees of reuse that are achieved by each analysis configuration. Artifacts refer to BugSwarm program pairs, for which we report final program size (in kLOC), program edit size (eLOC), and application-only callgraph size (|CG|). Then, we report for each analysis configuration the number of abstract states computed during reanalysis after applying the edit, in raw terms ($\#\varphi$) for batch analysis and as a percentage ($\%\varphi$) of the batch analysis baseline for each other configuration, as well as the amount of time required.

				Batch		Incremental		Demand-Driven		Demanded	
artifact	kLOC	eLOC	|CG|	$\#\varphi$	(s)	$\%\varphi$	(s)	$\%\varphi$	(s)	$\%\varphi$	(s)
BugSwarm 1	15.8	4	1599	17084	0.40	0.2	<0.01	48.7	0.27	<0.1	<0.01
BugSwarm 2	39.4	4	8212	77954	6.94	0.1	0.04	5.6	0.23	0.0	<0.01
BugSwarm 3	63.8	4	10048	100112	6.70	17.9	1.12	63.9	5.25	0.6	0.18
BugSwarm 4	23.5	4	2559	23377	0.73	0.4	0.01	2.3	0.02	0.3	<0.01
BugSwarm 5	36.2	8	7379	73322	6.88	<0.1	0.03	3.4	0.09	<0.1	<0.01
BugSwarm 6	45.7	134	3866	35395	1.35	<0.1	0.02	0.0	<0.01	0.0	<0.01
BugSwarm 7	32.3	4	1233	7029	0.20	12.5	0.02	28.0	0.10	0.2	<0.01
BugSwarm 8	147.6	56	2920	12199	0.25	1.1	0.01	0.3	<0.01	0.0	<0.01
BugSwarm 9	39.6	8	8260	78906	6.57	<0.1	0.03	2.4	0.07	<0.1	<0.01
BugSwarm 10	15.2	20	3558	44604	3.70	0.1	0.02	0.0	<0.01	0.0	<0.01
average	45.9	25	4963	46998	3.37	3.2	0.13	15.5	0.60	0.1	0.02

In order to resolve procedure calls, we rely on an upfront application-only callgraph computed using the WALA (WALA, 2021) analysis library. This allows us to support dynamic dispatch and formal/actual parameter binding rather than just parameter-less direct calls. WALA does not provide incremental call graphs, but techniques exist for incremental call-graph construction which we could adopt in the future.

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

The analysis engine is domain-agnostic: it is parameterized over an abstract domain module (Fig. 5.6) which provides standard abstract domain operations but can be implemented in a general-purpose language and needs not be aware of incrementality or demand.

We have successfully instantiated the implementation with interval and octagon numerical domains based on the APRON numerical domain library (Jeannet and Miné, 2009) and a simple list-segment shape analysis domain, all of which are of infinite height and have non-monotonic widenings.

6.4.2 Evaluation Corpus and Experimental Design

We run our prototype analysis framework on a subset of the BugSwarm dataset, which consists of program *pairs* drawn from open-source applications and their continuous integration histories (Tomassi et al., 2019). Each program pair consists of a "fail" version in which a CI failure was observed, and a subsequent "pass" version in which the failure is corrected. As such, they represent a natural use-case for a program analysis tool: after witnessing the failure, a user may analyze the program up to that point and verify the correctness of their fix.

We restrict our attention to pairs where the edit is in application code (rather than configuration or tests), affects between 1 and 200 lines of code, and the failure is not a compilation failure. On average, each program is 45 kLOC and consists of around 5000 distinct procedures. They make

extensive use of Java language features including exceptions and a wide variety of control-flow mechanisms, but we do not consider programs that make use of lambda expressions or method references.

Notably, every program in the corpus includes some recursive procedure(s) and is thus out of reach of the operational DAIG approach of Chapter 5.

In our experiments (shown in Table 6.1) we run an interval analysis (as in the overview example of Chapter 4) on ten Java program pairs from the BugSwarm dataset. This abstract domain is widely used in practice for numerical analysis problems, but is difficult to handle incrementally due to its infinite height and non-monotonic widening operator.

We consider four variants to re-analyze each program after an edit:

- *Batch* analysis, where the program is analyzed exhaustively from scratch in response to edits;
- *Incremental* analysis, which analyzes the program exhaustively and eagerly in response to edits, but reuses results from the previous version where possible;
- *Demand-Driven* analysis, which performs the minimal analysis work to respond to a client-issued query, but discards all facts in response to edits; and
- *Demanded* (i.e., incremental and demand-driven) analysis, which both reuses previous-version results and only performs the minimal analysis needed to respond to queries.

The four variants amount to configuration options in our analysis framework: essentially, the incremental analyses apply the edit semantics of Fig. 5.5 while the demand-driven analyses apply the query semantics of Fig. 5.4. For the demand-driven configurations, we select query locations that correspond to the failures observed in the initial program version or as near as possible in our internal representation.

The results are shown in Table 6.1, where we focus on symbolic metrics of analysis cost in terms of abstract states computed during reanalysis. We see that batch analysis requires computing tens of thousands of abstract states for each program edit. Both incremental analysis and demand-driven analysis represent significant improvements over the batch approach, but exhibit high variability:

depending on the location and relative proximity of queries, edits, and their dependencies, both approaches can incur significant analysis cost, on the order of 10% of that required for batch analysis on average . The combination of incrementality and demand, though, consistently outperforms both incremental-only and demand-driven-only analysis, requiring only 0.1% of the analysis cost of the baseline batch analysis and producing results in the interval domain in fractions of a second.

6.5 Conclusion

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

The key innovation in our framework is *demanded summarization*, which instantiates demanded abstract interpretation graphs (DAIGs) on demand to synthesize summaries as needed for a compositional interprocedural analysis, where a significant technical challenge is soundly handling self-referential summaries that naturally arise from demanded summarization of recursive procedures.

Our evaluation of a prototype implementation on real-world edits in Java programs provides evidence for realizing interactive abstract interpretation backed by compositional, summary-based analysis with arbitrarily complex abstract domains.

Chapter 7

Extensions and Variants of Demanded Summarization

The previous chapter describes *demanded summarization*: a technique for incremental and demand-driven abstract interpretation of recursive interprocedural programs in arbitrary abstract domains. We also showed that demanded summarization is not only sound but also from-scratch consistent, which is to say exactly as precise as the underlying batch abstract interpreter. Taken together, these features represent a significant advance over the previous state-of-the-art, both in terms of interactivity and expressivity.

However, the algorithm and formalism of Chapter 6 do not immediately yield a practical analysis tool nor fully realize the vision laid out in Chapter 2 of a real-time-interactive and general framework for program analysis. This chapter addresses some practical concerns that will arise in the implementation of an analysis framework based on demanded summarization graphs.

First, since our approach makes extensive use of caching and memoization, it must inevitably confront memory limitations. However, there is no means to do so in the DSG operational semantics of Section 6.2, which indiscriminately memoize all intermediate results. We develop and formalize in Section 7.1 some techniques to gracefully handle memory pressure in this chapter, then show that they preserve key metatheoretic guarantees of DSGs.

Also, though we fix a policy of maximizing precision in the previous chapter, it may be desirable to weaken or merge summaries during analysis to maximize summary reuse. We discuss in Section 7.2 some important design considerations to be made around summary reuse: when is a procedure summary "good enough" to apply, and what are the tradeoffs when weaker-than-needed

summary are permitted?

Lastly, the formalism of Section 6.3 reasons about a single analysis session that's initialized with an initial program and no cached analysis results. In practice, the ability to persist and deserialize summaries can make an analysis system more efficient and reliable, facilitate interoperation with batch compositional analyses, and enable analysis computation to be distributed across developer machines and cloud CI infrastructure. We describe in Section 7.3 how to extend the DSG framework to deal with precomputed summaries and persist them across analysis instances, noting the conditions that must be satisfied to do safely and showing that soundness and from-scratch consistency can be preserved.

7.1 Memory Pressure and Cache Invalidation

In a perfect world with infinite memory, we could simply store all previously-computed analysis results forever, keeping them on hand in case they are ever needed in the future. This is the world in which DSGs are defined and formalized in Chapter 6: DAIGs are instantiated and added to $\mathcal{D}^*$ by Q-Instantiate whenever they are needed to solve a query. Though their *contents* are dirtied in response to edits by the semantics of Fig. 6.5, DAIGs themselves are never disposed of. This was an intentional design decision to keep the presentation simple and focused on core contributions, but as a result a direct implementation of the operational semantics of Chapter 6 will clearly not yield a feasibly scalable analysis engine. This section extends the DSG formalism to explicitly account for memory-conserving operations, and shows that soundness, termination, and from-scratch consistency are all preserved under the extension.

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

Instead, we'd like to express that the analysis can choose to apply a $\mathcal{G} \rightsquigarrow \mathcal{G}'$ operation whenever it wants.

7.1.1 Discarding Summaries

The most direct way to handle memory pressure is simply by discarding procedure summary DAIGs that are not depended upon by others:

$$\textsc{Discard}$$
$$\frac{(\rho', \varphi') \leftsquigarrow (\rho^\ell, \varphi) \in \Delta \implies \Big(\rho' = \rho^\ell \wedge \varphi' = \varphi\Big) \qquad \Delta' = \Big\{\delta \Big| \delta = (\rho^\ell, \varphi) \leftsquigarrow (\_, \_) \in \Delta\Big\}}{\langle \mathcal{D}^*; (\rho^\ell, \varphi) \mapsto \mathcal{D}, \Delta\rangle \rightsquigarrow \langle \mathcal{D}^*, \Delta / \Delta'\rangle}$$

We use underscores in the premises to denote un-constrained metavariables, so this rule states that we can discard a summary DAIG $\mathcal{D}$ for a procedure ρ^ℓ with precondition φ if its only dependencies are self-referential (first premise), discarding any dependencies it may have held in the process (second premise).

Note that repeated applications of Discard can drop any or all DAIGs in a DSG, since the only cycles allowed in Δ are self-loops. That is, any DAIG can be discarded, so long as its transitive dependencies are all discarded first.

This operation plainly preserves DSG semantic consistency (Definition 6.1), as it only removes results with no non-self-referential dependencies.

Theorem 7.1 (Consistency Preservation under Discard)**.** *If $\mathcal{G} \rightsquigarrow \mathcal{G}'$ and $\mathcal{G}$ is semantically consistent with respect to P, then $\mathcal{G}'$ is also semantically consistent with respect to P.*

Proof. We'll consider the four bulleted conditions in sequence.

- Each constituent DAIG in $\mathcal{G}'$ is also in $\mathcal{G}$. Since it was consistent in $\mathcal{G}$ with the corresponding procedure CFG of P and abstract interpreter, it is also consistent in $\mathcal{G}'$.
- Due to the first premise of Discard, there are no dangling dependency edges in $\mathcal{G}'$ upon the removed DAIG.

- Due to the first and second premises of DISCARD, there are no dependency edges in $\mathcal{G}'$ that make any reference to the removed DAIG $\mathcal{D}^{\mathcal{G}}_{\rho,\varphi}$.
- The analysis results in $\mathcal{G}'$ are a strict subset of those in $\mathcal{G}$, so this universally-quantified condition is clearly preserved.

□

Since consistency is preserved under this operation, so too are the soundness, termination and from-scratch consistency results of Section 6.3.

This is of course an unsurprising result: dropping cached previously-computed results may require additional computation in the future to rebuild those results, but will never lead to incorrect results being computed. And, due to query termination, any necessary rebuilding of results can be done in finite time. Nonetheless, it is a valuable sanity check.

7.1.2 Condensing DAIGs to Summaries

Simply discarding entire DAIGs as described in the previous section is more heavy-handed than would be desired in many applications. In particular, note that DAIGs contain cached intermediate results at the granularity of statements/transfer functions, but summary application in DSGs operates at the granularity of procedures/summaries.

Instead of discarding an entire DAIG, an analyzer may exploit the fact that only its entry and exit abstract state are needed to summarize procedure calls, by caching those two states and discarding any intermediate results. Effectively, this technique allows us to reclaim the memory overhead of DAIGs at the cost of some incremental reuse whenever a summary is affected by an edit.

However, the definitions of Chapter 6 make no distinction between DAIGs and summaries; summaries are acquired only by the S-QUERY rule, which requires a corresponding fully-solved DAIG.

As such, we must extend the syntax and semantics of DSGs to express and reason about direct summarization. Formally, we first add a summary table $\mathcal{T}$ to the definition of demanded

summarization graphs $\mathcal{G}$, leaving the other components unchanged.

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

A summary table $\mathcal{T}$ is a collection of procedure summary Hoare triples $\{\varphi\}\, \rho\, \{\varphi'\}$ not backed by any DAIG in $\mathcal{D}^*$. For the most part, the DSG operational semantics (Figs. 6.4 and 6.5) "just work" with DSGs replaced by summary-table-equipped analogues and summary tables $\mathcal{T}$ threaded through the rules accordingly.

However, there are some additional rules needed to tabulate and apply summaries and some non-trivial tweaks required. As such, we reproduce the modified rules and provide the new rules in Fig. 7.1, highlighting the modifications relative to Chapter 6 in green.

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

Tabulate

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

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

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

The procedure summarization judgment $\mathcal{G} \vdash \{\varphi\}\, \rho\, \{\varphi'\} \,;\, \mathcal{G}'$ effectively abstracts away the two different types of cached results in our summary table-equipped DSGs: S-Query (Section 6.2) interprets fully-solved DAIGs as summaries, while S-Apply (Fig. 7.1) interprets rows of the summary table $\mathcal{T}$ as summaries.

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

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

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

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

Figure 7.1: Modifications and additions to demanded summarization query and edit semantics required to handle explicit summary tables $\mathcal{T}$. Additions are highlighted in green, and all rules *not* shown here are unchanged from Chapter 6. As before, underscores denote un-constrained metavariables.

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

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

Applying the Tabulate transformation allows a DSG-based analyzer to selectively *coarsen* its memoized results, achieving a significantly smaller memory footprint at the cost of fine-grained incremental reuse.

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

We will highlighht the key points but not reproduce the proofs in full here as they are largely unchanged from Chapter 6.

First, some modifications are required in the definition of DSG Semantic Consistency (Definition 6.1) to accomodate the summary triples in $\mathcal{T}$.

Definition 7.1 (DSG Semantic Consistency with Summary Tables)**.** *Conditions (1) and (2) of Definition 6.1 are unmodified and therefore elided here. Conditions (3) and (4) are modified and a condition (5) added as follows*

(3) Dependency edges $(\rho', \varphi') \overset{n}{\leftarrow} (\rho, \varphi)$ *in* Δ *are consistent with the relevant DAIGs and summary triples, such that whenever* $(\rho', \varphi) \in \mathsf{dom}(\mathcal{D}^*)$ *we have either*

- $\mathcal{D}^*(\rho', \varphi')(n) = \varphi_{\mathsf{callee\text{-}exit}}$ *(when* $\rho \neq \rho'$*) or*

- $\mathcal{D}^*(\rho', \varphi')(n) \sqsupseteq \varphi_{\mathsf{callee\text{-}exit}}$ *(when* $\rho = \rho'$*).*

where $\varphi_{\mathsf{callee\text{-}exit}} = \begin{cases} \varphi' & \text{if } \exists\ \{\varphi\}\ \rho\ \{\varphi'\} \in \mathcal{T} \\ \mathcal{D}^*(\rho, \varphi)(\mathsf{exit}(\rho)) & \text{otherwise} \end{cases}$

(4) Analysis results in DAIGs and summary triples are equal to the corresponding invariants produced by batch tabulation, where the domains coincide:

- $\mathcal{D}^{\mathcal{G}}_{\rho,\varphi}(\underline{\ell}) = I(\varphi, \ell)$ *for all* $\mathcal{D}^*(\rho, \varphi)(\underline{\ell})$ *where* $(\varphi, \ell) \in \mathsf{dom}(I)$*, and*
- $\varphi' = I(\varphi, \mathsf{exit}(\rho))$ *for all* $\{\varphi\}\ \rho\ \{\varphi'\} \in \mathcal{T}$ *where* $(\varphi, \mathsf{exit}(\rho)) \in \mathsf{dom}(I)$*.*

(5) For all $\{\varphi\}\rho\{\varphi'\} \in \mathcal{T}$*, we have that* $\Delta \supseteq \Delta'$ *where* Δ' *is defined by* $\langle \varepsilon, \varepsilon, \varepsilon \rangle \vdash_{\varphi} \mathsf{exit}(\rho) \Downarrow \varphi'; \langle \_, \Delta', \_ \rangle$

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

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

Theorem 7.2 (Consistency Preservation under Tabulate)**.** *If* $\mathcal{G} \rightsquigarrow \mathcal{G}'$ *and* $\mathcal{G}$ *is semantically consistent with respect to* P*, then* $\mathcal{G}'$ *is also semantically consistent with respect to* P*.*

Proof Sketch. Conditions (1) and (2) are vacuously preserved as in Theorem 7.1. Both modified conditions (3) and (4) are ensured by Tabulate and Definition 6.1 of $\mathcal{G}$, since the postcondition of a tabulated triple is exactly the exit abstract state of the discarded DAIG. Condition (5) is similarly guaranteed by the definition of Tabulate, where Δ contains all requisite dependnecies for $\mathcal{D}$ (by semantic consistency of $\mathcal{G}$) and is unchanged in $\mathcal{G}'$. □

Theorem 7.3 (Consistency Preservation)**.** *Semantic consistency of summary table-equipped DSGs is preserved under queries and edits as in Lemma 6.2.*

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

Theorem 7.4 (Termination)**.** *Queries and edits terminate in semantically consistent summary table-equipped demanded summarization graphs.*

Proof Sketch. The added rules S-Apply and D-Summary clearly terminate — S-Apply trivially, D-Summary by the same argument given for D-DAIG in Section 6.3. The modified rules Q-Instantiate and E-Delegate also terminate, both by the same arguments given originally in Section 6.3. □

Theorem 7.5 (From-Scratch Consistency and Soundness)**.** *Demanded analysis in summary table-equipped demanded summarization graphs produces from-scratch consistent (and therefore sound) results.*

Proof Sketch. The Tabulate operation only produces summaries that are identical to their DAIG analogues, which we know to be from-scratch consistent. Furthermore, E-Delegate and D-Summary ensure that any summary triple potentially affected by an edit is discarded. Therefore, summary triples are only applied under the same conditions as their DAIG analogues would have been, so the abstract semantics are identical to those of Chapter 6. □

Thus, the Tabulate operation presents a tradeoff between memory footprint and granularity of incremental reuse, without weakening the key features and guarantees of demanded summarization.

Both operations of the $\mathcal{G} \rightsquigarrow \mathcal{G}'$ judgment presented in this section — dropping DAIGs entirely with Discard and collapsing them down to input/output summaries with Tabulate — can be used to easily implement memory conservation mechanisms without jeopardizing hard-won guarantees of soundness, termination, and from-scratch consistency.

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

7.2 Summary Weakening for Reuse

In order to apply a summary at a procedure call, the core DSG operational semantics laid out in Chapter 6 require that its precondition *exactly* matches the call-site abstract state[1] . However, an analysis implementation may wish to maximize the reuse of previously computed summaries by applying a compatible one with a weaker-than-needed precondition instead.

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

S-Consequence

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

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

The primed extensions of Δ and $\mathcal{T}$ in the conclusion of S-Consequence serve to avoid a subtle issue: without them, when this rule is used to weaken a summary, the Q-Apply-Summary rule adds a dependency edge corresponding to the weakened (conclusion) summary rather than the stronger

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

(premise) summary from which it was derived. As a result, when the premise summary is dirtied the analysis would unsoundly fail to propagate that change to the callsite where its weakened form was applied. This shortcoming is fully addressed, though, by adding the derived triple to $\mathcal{T}$, along with a dependency edge in Δ on the (weaker) premise triple, since dirtying will transitively propagate the change through the derived triple $\{\varphi\}\,\rho\,\{\varphi''\}$ in $\mathcal{T}'$.[2]

Weakening summaries by applying the S-Consequence preserves the *soundness* of analysis results by the same reasoning that it is sound to apply the consequence rule in a standard Hoare logic. That is, the weaker triple $\{\varphi'\}\,\rho\,\{\varphi''\}$ in its premise implies the stronger triple inferred as its conclusion: since $\varphi \sqsubset \varphi'$, any concrete state modelling φ also models φ' and is thus guaranteed to be mapped by the semantics of ρ to a state modelling φ''.

However, introducing the S-Consequence rule comes at the cost of from-scratch consistency for demanded analysis. To see why, consider an edit immediately preceding a procedure call as illustrated in Fig. 7.2.

In the initial state, we have analyzed a procedure `foo` in the interval domain, using a summary of its callee `bar`. The `foo` procedure is a simple contrived example here: it just generates a random input integer `n` and then invokes `bar` on it.

When an edit is made on line 2, local analysis results in `foo` are dirtied but the $\{n \mapsto \top\}\texttt{bar}\{\varphi\}$ summary is not dirtied because its validity is not affected.

On the left, we can see the usefulness of the S-Consequence rule: this summary of `bar` can be reused to reanalyze `foo` without descending into its callees, because the summary's precondition $n \mapsto \top$ is weaker than the new abstract state $n \mapsto [0, \infty)$ at the line-3 callsite.

However, there is a catch: on the right, we reanalyze `foo` without the rule of consequence and get a potentially different and more precise result φ'. Crucially, this result is what would be computed using the demanded summarization semantics of Chapter 6 and is therefore from-scratch consistent with batch tabulation.

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

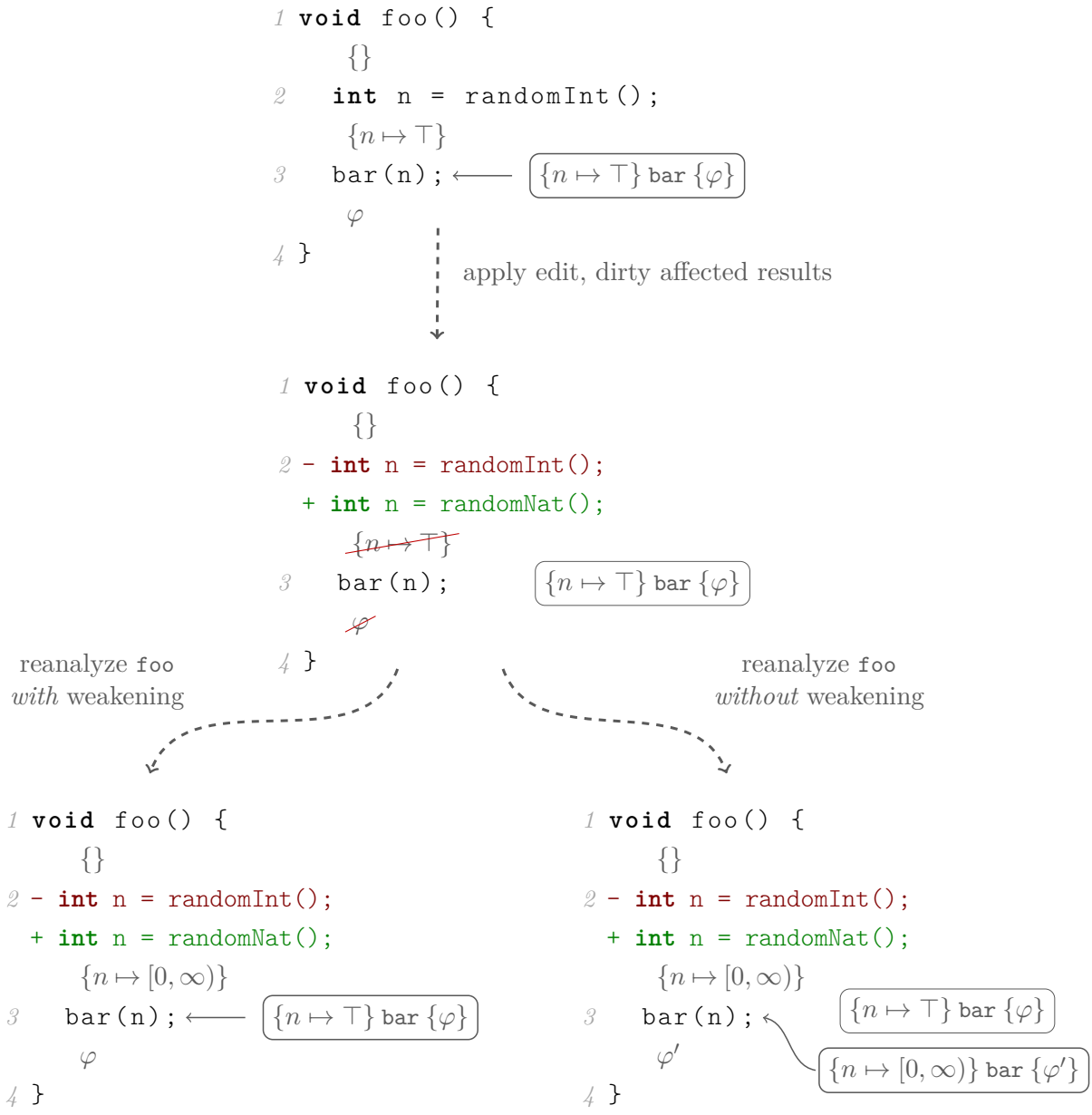


Figure 7.2: A simple example designed to highlight the effect of the S-Consequence "weakening" rule on summary reuse and from-scratch consistency.
In the initial state, `foo` has been fully analyzed using a summary of `bar`. An edit to `foo` at line 2, just before the call to `bar`, dirties downstream intraprocedural results but not the `bar` summary.
On the left, we can soundly reapply the original summary using S-Consequence because its precondition is strictly weaker than the new callsite abstract state.
On the right, without S-Consequence, we must instead compute a new summary of `bar`, with precondition exactly the new callsite abstract state.

Intuitively, this is because the cached summary $\{n \mapsto \top\}$ `bar` $\{\varphi\}$, though it is sound and precise with respect to its precondition, is not necessarily the same summary that would be computed by a from-scratch batch tabulation. In this case, a post-edit batch tabulation would just compute the $\{n \mapsto [0, \infty)\}$ `bar` $\{\varphi'\}$ partial summary of `bar`.

So, allowing summaries to be weakened using S-Consequence violates from-scratch consistency but preserves soundness and termination. As such, it represents a tradeoff and a tuneable parameter in the design of practical interactive analysis tools based on demanded summarization. In some circumstance, the benefit (computational savings due to increased summary reuse) may be well worth the cost of weaker precision guarantees; in others, the predictable behavior and maximal precision of a from-scratch consistent demanded analysis are more important.

7.3 Persisting Summaries Across Analysis Instances

Up to this point, we have considered demanded summarization graphs as self-contained: analysis begins with a program and an empty cache, then computes and/or discards analysis facts in response to client-issued queries and edits. This serves to simplify the presentation and discussion, but in practice there are several reasons it may be useful to either instantiate a demanded summarization graph with a pre-populated cache of summaries or persist summaries across analysis instances.

For example, it is inevitable that the analysis or host machine will restart at some point; by persisting/reading analysis results to/from disk, an analyzer can avoid repeating analysis with each such interruption and provide initial analysis results quickly with a warm start.

A local DSG-based analysis engine may also wish to make use of procedure summaries computed in powerful cloud infrastructure, analyzing locally only what is needed to update results and respond to queries during the development process. This form of interoperability between demanded and batch summary-based analysis, as depicted in Fig. 7.3 and alluded to in Chapter 2, is crucial due to the ubiquity and power of batch compositional analysis in real-world analysis deployments (Calcagno et al., 2011; Fähndrich and Logozzo, 2010).

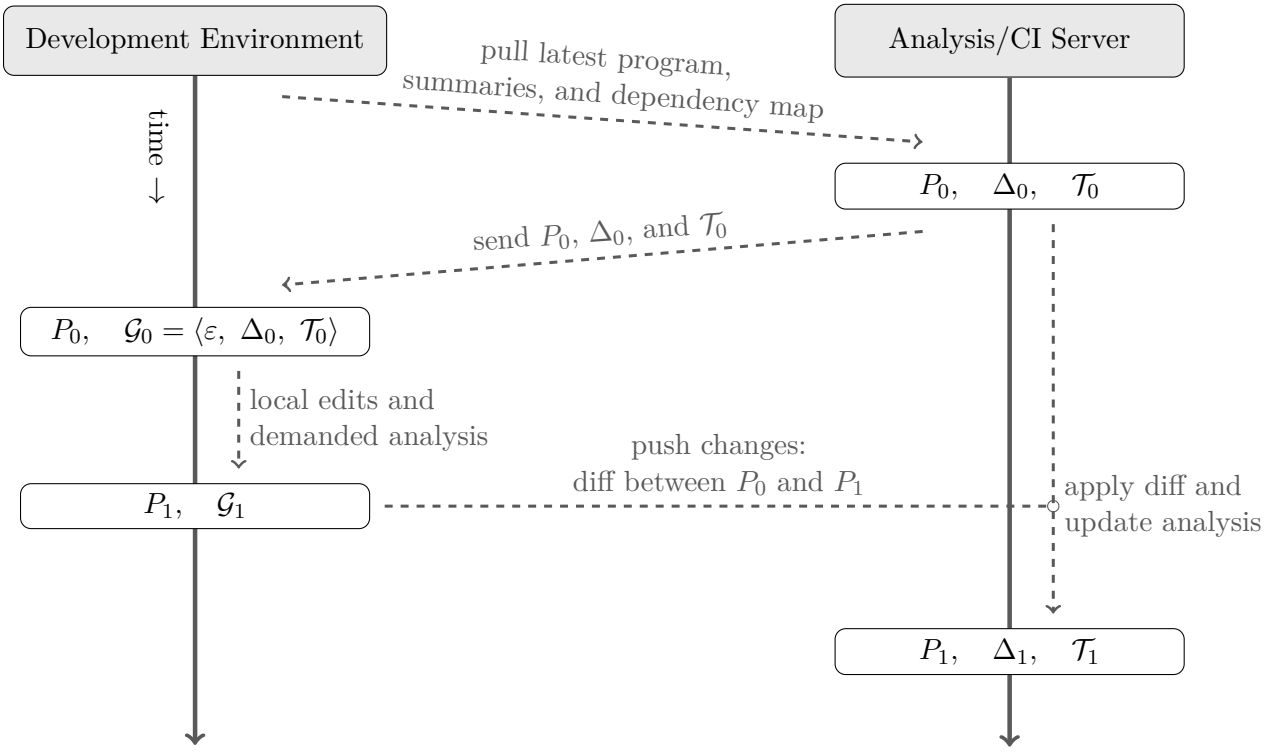


Figure 7.3: Interoperation between demanded summarization (in a development environment) batch tabulation (in a continuous integration environment). The developer pulls some procedure summaries $\mathcal{T}_0$ (and the dependencies Δ_0 among them) along with the initial program P_0. Then, they edit the program and query a local analysis engine, eventually producing some updated program P_1 and pushing their changes to the central server. The CI server can then update its summary database to $\mathcal{T}_1$ and Δ_1 using either a batch tabulation engine or DSG-based demanded analysis, and the process continues.

Fig. 7.3 shows an example of such interaction between a developer and some remote batch analysis infrastructure. When they begin working on some changes to a program P, the developer pulls the current version P_0 along with some procedure summaries $\mathcal{T}_0$ and dependency map Δ_0.

Using $\mathcal{T}_0$ and Δ_0, the developer initializes a DSG $\mathcal{G}_0$, supporting them with real-time-interactive analysis results as they edit P_0 and eventually produce some updated program P_1. They then push their changes to the central server, which can update its summary table and dependencies to $\mathcal{T}_1$ and Δ_1 respectively using either a DSG-based system for demanded analysis or simply a batch summary-based analysis algorithm. This process now continues, with powerful analysis infrastructure keeping a collection of summaries in support of local demanded analyses providing real-time interaction. Modulo the usual (orthogonal) concerns around merge conflicts, such a system could potentially accomodate multiple developers interacting with the shared analysis infrastructure.

Metatheory. Though it is a form of demanded summarization, the analysis design described in the previous section has some key differences from the system formalized and implemented in Chapter 6. As such, it is not obvious which characteristics and formal guarantees it inherits under which conditions.

Consider the DSG $\mathcal{G}_0 = \langle \varepsilon,\ \Delta_0,\ \mathcal{T}_0 \rangle$ for a program P_0, composed of no DAIGs, a non-empty summary table $\mathcal{T}_0$ and a dependency map Δ_0. Intuitively, $\mathcal{G}_0$ is an initial DSG for analysis of P_0, with a pre-populated table $\mathcal{T}_0$ of partial procedure summaries and dependencies Δ_0 thereof. Due to Theorem 7.4 and Theorem 7.5, we know that demanded analysis from this initial analysis state is sound, terminating, and from-scratch consistent so long as $\mathcal{G}_0$ is semantically consistent (Definition 7.1).

$\mathcal{G}_0$ satisfies consistency conditions (1), (2), (3), and the first bullet point of (4) vacuously, since $\mathcal{D}^* = \varepsilon$. Thus, we require Δ_0 and $\mathcal{T}_0$ to satisfy the second bullet point of condition (4) and condition (5), respectively stating that $\mathcal{T}_0$ agrees with batch tabulation and that Δ_0 over-approximates the semantic dependencies of every triple in $\mathcal{T}_0$

A summary table $\mathcal{T}_0$ that agrees with batch tabulation can be computed either by simply using a batch tabulation solver and serializing its results or some subset thereof, or by applying

TABULATE to some or all DAIGs in a demanded summarization graph.

On the other hand, ensuring that Δ_0 over-approximates the dependencies among summary triples in $\mathcal{T}_0$ is less simple. If a DSG is used to compute the summaries, then its dependency map can be persisted alongside the summary table for future reuse.

However, if a batch tabulation solver or other system is used to compute $\mathcal{T}_0$, then no such Δ_0 is immediately available. In this case, either (a) the solver can be instrumented to output an interprocedural dependency map by logging whenever a summary is applied, or (b) an overapproximation can be built from the program's callgraph.

A callgraph-based dependency map is less precise than the dependency map Δ maintained by a DSG, though, and may lead to needlessly conservative dirtying of procedure summaries when a semantically-irrelevant callee summary is dirtied — for example, if a summary of `foo` depends on one summary of `bar` and a different summary of `bar` is dirtied, or if `foo` has an unreachable call to `bar`.

Discussion. The tweak to the DSG system described here — initiating interactive analysis with some pre-populated summaries and dependencies rather than an empty cache — may seem minor, but it goes a long way towards enabling our vision for interactive abstract interpretation. In particular, it facilitates interoperation with non-incremental compositional analyses, which are commonly used in practice (Calcagno et al., 2011; Fähndrich and Logozzo, 2010) and widely seen as essential for scaling sophisticated analyses up to large code bases (Distefano et al., 2019; Sadowski et al., 2018).

That is, this small extension to the base DSG framework of Chapter 6 allows for summaries to computed in powerful cloud infrastructure or in a distributed manner, then supplied to a local development environment to facilitate real-time interaction with analysis tools as a programmer makes edits and issues queries. What's more, the interactively-computed analysis results are exactly as precise as what would be computed from-scratch by a costly full re-analysis, due to the careful design of our demanded summarization and dependency tracking algorithms.

Chapter 8

Conclusions and Future Directions

In this dissertation, we have developed techniques for *incremental* and *demand-driven* abstract interpretation, designed to efficiently support real-time developer interactions with analysis tools. Our approach works by tracking the semantic dependencies among intermediate analysis results, dynamically updating dependency structures to soundly handle the complex and unbounded fixed-point computations inherent to abstract interpretation. It leverages insights from the incremental computation community which point towards the *combination* of incrementality and demand as an elegant and efficient tool for interactive system design, while preserving the generality and core metatheoretic guarantees of the abstract interpretation framework.

We first presented a framework for demanded *intraprocedural* abstract interpretation (Chapter 5), in which fixed-point computations over loops can be unrolled in demanded abstract interpretation graphs (or DAIGs) — the first technique for incremental and demand-driven abstract interpretation in arbitrary abstract domains. Then, we built a demanded *interprocedural* analysis using DAIGs as building blocks to produce compositional procedure summaries and dynamically updating interprocedural dependencies at analysis-time to analyze recursive programs incrementally and on-demand (Chapter 6). This technique preserves the metatheoretic guarantees and domain agnosticity of our intraprocedural analysis, but handles a much wider range of real-world programs and control-flow structures. We also developed a range of extensions and variations of demanded summarization (Chapter 7), providing formal underpinnings for a key handful of optimizations and practical implementation details for deployments of interactive analysis tools — showing how to

manage memory and computation constraints without sacrificing hard-won theoretical results of soundness, termination, and from-scratch consistency.

8.1 Future Directions

This dissertation aims to set out a path for the development of a new generation of analysis tools designed from the ground up for developer experience and interactivity. We briefly discuss here some research directions that we see as promising and logical future work towards this goal.

Varied Abstract Interpretation Algorithms. There are many variations of abstract interpretation, designed to analyze different types of program, prove different properties, or achieve different performance characteristics. For clarity of presentation and design, we focused in this dissertation on some generic, bog-standard techniques for forwards abstract interpretation with widening and tabulation. However, more precision or better performance may be achieved with more sophisticated algorithms.

For example, we fix a *widening strategy* of applying ∇ at every abstract iteration until a fixed point is reached (Stein et al., 2021a), but it is possible (and common in practice) to instead widen only intermittently or after some threshold of joins fails to converge, or to analyze to a pre-fixed point rather than a fixed point (Bourdoncle, 1993; Cousot and Cousot, 1977).

We also focus on the abstract interpretation over the control flow of imperative programs – their control flow and call graphs. Abstract interpretation-based *sparse* analyses, which operate over program data flow (i.e. def/use) graphs instead, may also benefit from demanded evaluation techniques analogous to those developed in this dissertation (Oh et al., 2012; Tok et al., 2006).

Similarly, some abstract interpreters analyze the program's control flow in reverse – such *backwards* abstract interpreters can be used for precondition inference (Cousot et al., 2013), to refute false alarms (Blackshear et al., 2013; Miné, 2014), or to refine another analysis' abstract state (Stein et al., 2019). Such techniques may be particularly amenable to demanded evaluation due to the query-driven nature of their applications.

Refinement. 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 ("abstraction refinement", e.g. (Henzinger et al., 2002; Gulavani and Rajamani, 2006; Oh et al., 2014; Andreasen and Møller, 2014)) or by incorporating information from some oracle ("value refinement", e.g. (Stein et al., 2019; Bodden et al., 2011; Dufour et al., 2007)).

Abstraction refinement has potential to improve precision and scalability in demanded abstract interpretation. On the other hand, the precise dependency tracking of our technique could make value refinement more efficient and interactive for dynamic language developers, with extensions to the DSG framework to support extrinsically-provided analysis facts and update analysis results accordingly.

IDE Integrations and User Studies. Our focus in this dissertation has been on the formal specification and design of an interactive analysis engine, aiming to provide a solid formal foundation on which practical analysis tools can be built without ad-hoc hacks for interactivity. While this is a valuable contribution in its own right, it is not yet validated by real-world usage in developer workflows. By building out integrations with IDEs and investigating user interfaces for analysis interaction, interdisciplinary research with human-computer interaction and/or software engineering experts could shed a light on possible blind spots in our approach and point towards other future directions. This would be helpful not only for our demanded abstract interpretation but also a wide range of other incremental program analysis techniques, many of which rely on synthetic edits or version control histories to validate their approaches (Arzt and Bodden, 2014; Szabó et al., 2018, 2021; Liu et al., 2019; Söderberg and Hedin, 2012).

Bibliography

Martín Abadi, Butler W. Lampson, and Jean-Jacques Lévy. Analysis and Caching of Dependencies. In *International Conference on Functional Programming (ICFP)*, 1996.

Umut A. Acar, Guy E. Blelloch, and Robert Harper. Adaptive Functional Programming. In *Principles of Programming Languages (POPL)*, 2002.

Umut A. Acar, Amal Ahmed, and Matthias Blume. Imperative Self-Adjusting Computation. In *Principles of Programming Languages (POPL)*, 2008.

Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. 2006.

Esben Andreasen and Anders Møller. Determinacy in Static Analysis for jQuery. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2014.

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

Wayne A. Babich and Mehdi Jazayeri. The Method of Attributes for Data Flow Analysis: Part II. Demand Analysis. *Acta Informatica*, (3), 1978.

Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular Automatic Assertion Checking with Separation Logic. In *Formal Methods for Components and Objects (FMCO)*, 2005.

Sam Blackshear, Bor-Yuh Evan Chang, and Manu Sridharan. Thresher: Precise Refutations for Heap Reachability. In *Programming Language Design and Implementation (PLDI)*, 2013.

Sam Blackshear, Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. RacerD: Compositional Static Race Detection. *Proc. ACM Program. Lang.*, (OOPSLA), 2018.

Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A Static Analyzer for Large Safety-Critical Software. In *Programming Language Design and Implementation (PLDI)*, 2003.

Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming Reflection: Aiding Static Analysis in the Presence of Reflection and Custom Class Loaders. In *International Conference on Software Engineering (ICSE)*, 2011.

Rastislav Bodík, Rajiv Gupta, and Vivek Sarkar. ABCD: eliminating array bounds checks on demand. In *Programming Language Design and Implementation (PLDI)*, 2000.

François Bourdoncle. Efficient Chaotic Iteration Strategies with Widenings. In *Formal Methods in Programming and Their Applications*, 1993.

Cristiano Calcagno and Dino Distefano. Infer: An Automatic Program Verifier for Memory Safety of C Programs. In *NASA Formal Methods (NFM)*, 2011.

Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional Shape Analysis by Means of Bi-Abduction. *J. ACM*, (6), 2011.

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

Sagar Chaki, Edmund M. Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular Verification of Software Components in C. *IEEE Trans. Software Eng.*, (6), 2004.

Bor-Yuh Evan Chang and Xavier Rival. Relational Inductive Shape Analysis. In *Principles of Programming Languages (POPL)*, 2008.

Bor-Yuh Evan Chang, Xavier Rival, and George C. Necula. Shape Analysis with Structural Invariant Checkers. In *SAS*, 2007.

Yan Chen, Umut A. Acar, and Kanat Tangwongsan. Functional Programming for Dynamic and Large Data with Self-Adjusting Computation. In *International Conference on Functional Programming (ICFP)*, 2014a.

Yan Chen, Jana Dunfield, Matthew A. Hammer, and Umut A. Acar. Implicit Self-Adjusting Computation for Purely Functional Programs. *J. Funct. Program.*, (1), 2014b.

Patrick Cousot. Asynchronous iterative methods for solving a fixpoint system of monotone equations. Research Report IMAG-RR-88, Université Scientifique et Médicale de Grenoble, 1977.

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

Patrick Cousot and Radhia Cousot. Systematic Design of Program Analysis Frameworks. In *Principles of Programming Languages (POPL)*, 1979.

Patrick Cousot and Radhia Cousot. Modular Static Program Analysis. In *Compiler Construction (CC)*, 2002.

Patrick Cousot and Nicolas Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. In *Principles of Programming Languages (POPL)*, 1978.

Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTREÉ Analyzer. In *European Symposium on Programming (ESOP)*, 2005.

Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, and Xavier Rival. Why Does Astrée Scale Up? *Formal Methods Syst. Des.*, (3), 2009.

Patrick Cousot, Radhia Cousot, and Laurent Mauborgne. The Reduced Product of Abstract Domains and the Combination of Decision Procedures. In *FoSSaCS*, 2011.

Patrick Cousot, Radhia Cousot, Manuel Fähndrich, and Francesco Logozzo. Automatic Inference of Necessary Preconditions. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2013.

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

Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. A Local Shape Analysis Based on Separation Logic. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2006.

Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. Scaling Static Analyses at Facebook. *Commun. ACM*, (8), 2019.

Lisa Nguyen Quang Do, Karim Ali, Benjamin Livshits, Eric Bodden, Justin Smith, and Emerson R. Murphy-Hill. Just-in-Time Static Analysis. In *Software Testing and Analysis (ISSTA)*, 2017.

Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. Demand-Driven Computation of Interprocedural Data Flow. In *Principles of Programming Languages (POPL)*, 1995.

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

Bruno Dufour, Barbara G. Ryder, and Gary Sevitsky. Blended Analysis for Performance Understanding of Framework-Based Applications. In *Software Testing and Analysis (ISSTA)*, 2007.

Max Brunsfeld et al. tree-sitter/tree-sitter: v0.20.0, 2021.

Manuel Fähndrich and Francesco Logozzo. Static Contract Checking with Abstract Interpretation. In *Formal Verification of Object-Oriented Software (FoVeOOS)*, 2010.

Rodney Farrow. Automatic Generation of Fixed-Point-Finding Evaluators for Circular, but Well-Defined, Attribute Grammars. In *Compiler Construction (CC)*, 1986.

John Field and Tim Teitelbaum. Incremental Reduction in the Lambda Calculus. In *LISP and Functional Programming*, 1990.

Eric Goubault and Sylvie Putot. A Zonotopic Framework for Functional Abstractions. *Formal Methods Syst. Des.*, (3), 2015.

Eric Goubault, Sylvie Putot, and Franck Védrine. Modular Static Analysis with Zonotopes. In *Static Analysis (SAS)*, 2012.

Bhargav S. Gulavani and Sriram K. Rajamani. Counterexample Driven Refinement for Abstract Interpretation. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2006.

Sumit Gulwani and Ashish Tiwari. Combining Abstract Interpreters. In *Programming Language Design and Implementation (PLDI)*, 2006.

Matthew A. Hammer, Umut A. Acar, and Yan Chen. CEAL: a C-based language for self-adjusting computation. In *Programming Language Design and Implementation (PLDI)*, 2009.

Matthew A. Hammer, Georg Neis, Yan Chen, and Umut A. Acar. Self-Adjusting Stack Machines. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2011.

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

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

Nevin Heintze and Olivier Tardieu. Demand-Driven Pointer Analysis. In *Programming Language Design and Implementation (PLDI)*, 2001.

Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy Abstraction. In *Principles of Programming Languages (POPL)*, 2002.

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

David Van Horn and Matthew Might. Abstracting Abstract Machines. In *International Conference on Functional Programming (ICFP)*, 2010.

Susan Horwitz, Thomas W. Reps, and Shmuel Sagiv. Demand Interprocedural Dataflow Analysis. In *Foundations of Software Engineering (FSE)*, 1995.

Bertrand Jeannet and Antoine Miné. Apron: A Library of Numerical Abstract Domains for Static Analysis. In *Computer-Aided Verification (CAV)*, 2009.

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

John B. Kam and Jeffrey D. Ullman. Global Data Flow Analysis and Iterative Algorithms. *J. ACM*, (1), 1976.

John B. Kam and Jeffrey D. Ullman. Monotone Data Flow Analysis Frameworks. *Acta Inf.*, (3), 1977.

Gary A. Kildall. A Unified Approach to Global Program Optimization. In *Principles of Programming Languages (POPL)*, 1973.

K. Rustan M. Leino and Valentin Wüstholz. Fine-Grained Caching of Verification Results. In *Computer-Aided Verification (CAV)*, 2015.

Bozhen Liu, Jeff Huang, and Lawrence Rauchwerger. Rethinking Incremental and Parallel Pointer Analysis. *ACM Trans. Program. Lang. Syst.*, (1), 2019.

Magnus Madsen, Ming-Ho Yee, and Ondrej Lhoták. From Datalog to Flix: a Declarative Language for Fixed Points on Lattices. In *Programming Language Design and Implementation (PLDI)*, 2016.

Stephen Magill, Aleksandar Nanevski, Edmund Clarke, and Peter Lee. Inferring Invariants in Separation Logic for Imperative List-Processing Programs. In *Semantics, Program Analysis, and Computing Environments for Memory Management (SPACE)*, 2006.

Eva Magnusson and Görel Hedin. Circular Reference Attributed Grammars - their Evaluation and Applications. *Sci. Comput. Program.*, (1), 2007.

BugSwarm Maintainers. *SpigotMC-BungeeCord-130330788*, 2021a. `https://bugswarm.org/dataset`.

BugSwarm Maintainers. *tananaev-traccar-164537301*, 2021b. `https://bugswarm.org/dataset`.

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

BugSwarm Maintainers. *tananaev-traccar-188473749*, 2021d. `https://bugswarm.org/dataset`.

BugSwarm Maintainers. *tananaev-traccar-191125671*, 2021e. `https://bugswarm.org/dataset`.

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

BugSwarm Maintainers. *tananaev-traccar-255051211*, 2021g. `https://bugswarm.org/dataset`.

BugSwarm Maintainers. *vkostyukov-la4j-45524419*, 2021h. `https://bugswarm.org/dataset`.

BugSwarm Maintainers. *tananaev-traccar-64783123*, 2021i. `https://bugswarm.org/dataset`.

BugSwarm Maintainers. *square-okhttp-95014919*, 2021j. `https://bugswarm.org/dataset`.

Matthew Might, Yannis Smaragdakis, and David Van Horn. Resolving and Exploiting the k-CFA paradox: Illuminating Functional vs. Object-Oriented Program Analysis. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, 2010.

Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized Object Sensitivity for Points-to Analysis for Java. *ACM Trans. Softw. Eng. Methodol.*, (1), 2005.

Antoine Miné. The Octagon Abstract Domain. *High. Order Symb. Comput.*, (1), 2006.

Antoine Miné. Backward Under-Approximations in Numeric Abstract Domains to Automatically Infer Sufficient Program Conditions. *Sci. Comput. Program.*, 2014.

Greg Nelson and Derek C. Oppen. Simplification by Cooperating Decision Procedures. *ACM Trans. Program. Lang. Syst.*, (2), 1979.

Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, and Kwangkeun Yi. Design and Implementation of Sparse Global Analyses for C-like Languages. In *Programming Language Design and Implementation (PLDI)*, 2012.

Hakjoo Oh, Wonchan Lee, Kihong Heo, Hongseok Yang, and Kwangkeun Yi. Selective Context-Sensitivity Guided by Impact Pre-Analysis. In *Programming Language Design and Implementation (PLDI)*, 2014.

Lori L. Pollock and Mary Lou Soffa. An Incremental Version of Iterative Data Flow Analysis. *IEEE Trans. Software Eng.*, (12), 1989.

William Pugh and Tim Teitelbaum. Incremental Computation via Function Caching. In *Principles of Programming Languages (POPL)*, 1989.

Thomas Reps. Program Analysis via Graph Reachability. *Information and Software Technology*, (11-12), 1998.

Thomas W. Reps. Optimal-Time Incremental Semantic Analysis for Syntax-Directed Editors. In *Principles of Programming Languages (POPL)*, 1982.

Thomas W. Reps. Solving Demand Versions of Interprocedural Analysis Problems. In *Compiler Construction (CC)*, 1994.

Thomas W. Reps, Tim Teitelbaum, and Alan J. Demers. Incremental Context-Dependent Analysis for Language-Based Editors. *ACM Trans. Program. Lang. Syst.*, (3), 1983.

Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Principles of Programming Languages (POPL)*, 1995.

John C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Logic in Computer Science (LICS)*, 2002.

Noam Rinetzky and Shmuel Sagiv. Interprocedural Shape Analysis for Recursive Programs. In *Compiler Construction (CC)*, 2001.

Barbara G. Ryder. Incremental Data Flow Analysis. In *Principles of Programming Languages (POPL)*, 1983.

Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. Lessons from Building Static Analysis Tools at Google. *Commun. ACM*, (4), 2018.

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

Mauricio Santos. *Buckets-JS: A JavaScript Data Structure Library*, 2016. `https://github.com/mauriciosantos/Buckets-JS`.

Todd Sedano, Paul Ralph, and Cécile Péraire. Software Development Waste. In *International Conference on Software Engineering (ICSE)*, 2017.

Micha Sharir and Amir Pnueli. Two Approaches to Interprocedural Data Flow Analysis. In *Program Flow Analysis: Theory and Applications*. 1981.

Olin Shivers. Control-Flow Analysis in Scheme. In *Programming Language Design and Implementation (PLDI)*, 1988.

Gagandeep Singh, Markus Püschel, and Martin T. Vechev. Fast Polyhedra Abstract Domain. In *Principles of Programming Languages (POPL)*, 2017.

Emma Söderberg and Görel Hedin. Incremental Evaluation of Reference Attribute Grammars using Dynamic Dependency Tracking. (LU-CS-TR:2012-249), 2012.

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

Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. Demand-Driven Points-To Analysis for Java. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2005.

Benno Stein, Benjamin Barslev Nielsen, Bor-Yuh Evan Chang, and Anders Møller. Static Analysis with Demand-Driven Value Refinement. *Proc. ACM Program. Lang.*, (OOPSLA), 2019.

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

Benno Stein, Bor-Yuh Evan Chang, and Manu Sridharan. *Demanded Abstract Interpretation (artifact)*, 2021b. `https://doi.org/10.5281/zenodo.4663292`.

Benno Stein, Bor-Yuh Evan Chang, and Manu Sridharan. *DAI: Demanded Abstract Interpretation*, 2021c. `https://github.com/cuplv/dai`.

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

Tamás Szabó, Gábor Bergmann, Sebastian Erdweg, and Markus Voelter. Incrementalizing Lattice-based Program Analyses in Datalog. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2018.

Tamás Szabó, Sebastian Erdweg, and Gábor Bergmann. Incremental Whole-Program Analysis in Datalog with Lattices. In *Programming Language Design and Implementation (PLDI)*, 2021.

Alfred Tarski. A Lattice-Theoretical Fixpoint Theorem and its Applications. *Pacific Journal of Mathematics*, (2), 1955.

Teck Bok Tok, Samuel Z. Guyer, and Calvin Lin. Efficient Flow-Sensitive Interprocedural Data-Flow Analysis in the Presence of Pointers. In *Compiler Construction (CC)*, 2006.

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

Alexey Tregubov, Barry W. Boehm, Natalia Rodchenko, and Jo Ann Lane. Impact of Task Switching and Work Interruptions on Software Development Processes. 2017.

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

Greta Yorsh, Eran Yahav, and Satish Chandra. Generating Precise and Concise Procedure Summaries. In *Principles of Programming Languages (POPL)*, 2008.

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

F. Kenneth Zadeck. Incremental Data Flow Analysis in a Structured Program Editor. In *Compiler Construction (CC)*, 1984.