ABSTRACT

Title of dissertation: MECHANIZING ABSTRACT INTERPRETATION

David Darais
Doctor of Philosophy, 2017

Dissertation directed by: Professor David Van Horn
Department of Computer Science

It is important when developing software to verify the absence of undesirable behavior such as crashes, bugs and security vulnerabilities. Some settings require high assurance in verification results, *e.g.*, for embedded software in automobiles or airplanes. To achieve high assurance in these verification results, formal methods are used to automatically construct or check proofs of their correctness. However, achieving high assurance for program analysis results is challenging, and current methods are ill suited for both complex critical domains and mainstream use.

To verify the correctness of software we consider *program analyzers*—automated tools which detect software defects—and to achieve high assurance in verification results we consider *mechanized verification*—a rigorous process for establishing the correctness of program analyzers via computer-checked proofs.

The key challenges to designing verified program analyzers are: (1) achieving an analyzer *design* for a given programming language and correctness property; (2) achieving an *implementation* for the design; and (3) achieving a *mechanized*

verification that the implementation is correct w.r.t. the design. The state of the art in (1) and (2) is to use *abstract interpretation*: a guiding mathematical framework for systematically constructing analyzers directly from programming language semantics. However, achieving (3) in the presence of abstract interpretation has remained an open problem since the late 1990's. Furthermore, even the state-of-the art which achieves (3) in the absence of abstract interpretation suffers from the inability to be reused in the presence of new analyzer designs or programming language features.

First, we solve the open problem which has prevented the combination of abstract interpretation (and in particular, *calculational* abstract interpretation) with mechanized verification, which advances the state of the art in designing, implementing, and verifying analyzers for critical software. We do this through a new mathematical framework *Constructive Galois Connections* which supports synthesizing specifications for program analyzers, calculating implementations from these induced specifications, and is amenable to mechanized verification.

Finally, we introduce reusable components for implementing analyzers for a wide range of designs and semantics. We do this though two new frameworks *Galois Transformers* and *Definitional Abstract Interpreters*. These frameworks tightly couple analyzer design decisions, implementation fragments, and verification properties into compositional components which are (target) programming-language independent and amenable to mechanized verification. Variations in the analysis design are then recovered by simply re-assembling the combination of components. Using this framework, sophisticated program analyzers can be assembled by non-experts, and the result are guaranteed to be verified by construction.

MECHANIZING ABSTRACT INTERPRETATION

by

David Darais

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2017

Advisory Committee:
Professor David Van Horn, Chair/Advisor
Professor Patrick Cousot
Professor Jeff Foster
Professor Michael Hicks
Professor Larry Washington

Preface

Much of the material in this thesis has previously appeared in the following peer-reviewed publications, authored jointly with David Van Horn, Matthew Might, Nicholas Labich and Phúc C. Nguyễn:

> David Darais, Matthew Might, and David Van Horn. Galois transformers and modular abstract interpreters: Reusable metatheory for program analysis. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. ACM, New York, NY, USA, 2015
>
> David Darais and David Van Horn. Constructive Galois connections: Taming the Galois connection framework for mechanized metatheory. In *International Conference on Functional Programming (ICFP)*. ACM, New York, NY, USA, 2016
>
> David Darais, Nicholas Labich, Phúc C. Nguyễn, and David Van Horn. Definitional abstract interpreters for higher-order programming languages. In *International Conference on Functional Programming (ICFP)*. ACM, New York, NY, USA, 2017

Dedicated to my grandparents.

Alexander and Norma Darais

Byron and Ingrid Forsyth

Bob and Doris Edmondson

Louis and Doris Miner

Bob and Joyce Dustman

Acknowledgments

Thanks to my advisor David Van Horn for being such an amazing mentor, and for helping me every step of the way, even before I became his student. Thanks to Matt Might for being such an amazing collaborator and role model. Thanks to Mike Hicks, Jeff Foster and Nate Foster for their continued encouragement and support. Thanks to Éric Tanter and Ron Garcia for many helpful discussions of their work, as well as their warm encouragement ever since our meeting at OOPSLA '15. Thanks to Matthias Felleisen for giving me key advice at a pivotal moment during my PhD. Thanks to Patrick Cousot for many detailed and insightful comments on this thesis.

Thanks to my partner and love of my life Olivia, for always believing in me and taking care of me. Thanks to my parents—Tom and Suzanne Darais, and Karin and Jay Larson—for their unwavering love and support. Thanks to my uncle Steve for helping take care of me throughout my PhD studies, and for always being so loving, wise and amazing. Thanks to my brothers Abraham and Jeremiah Darais and my good friend Simon Williams for always being there for me. Thanks to my boston fam: Guillaume Basse, Omar Shammas, John Coglianese, Yazan Abu Ghazal, Dan Huang, Dan King, Scott Moore and Andrew Johnson; you never stopped supporting me, believing in me and cheering me on. Thanks to Kris Micinski for being such a solid friend ever since I moved to Maryland.

Thanks to those who peer-reviewed my scholarly submissions and provided helpful feedback, regardless of the acceptance outcome. And finally, thanks to those who I forgot to mention who have undoubtedly helped me along the way.

Table of Contents

List of Figures

Chapter 1: Introduction

This thesis aims to improve the correctness and reliability of software. The results from this thesis offer methods to cost-effectively prevent software failures and exploits, and reduce their costs on society. The methods we develop are a new mathematical theory which improves the state-of-the art in foundational approaches to software reliability, and two new program analysis frameworks which help reduce the cost of achieving software reliability. These contributions support the following thesis: *Constructing mechanically verified program analyzers via calculation and composition is feasible using constructive Galois connections and modular abstract interpreters.*

The Software Reliability Problem Software bugs are expensive. Software plays an important role in *critical* systems like automobiles, aircraft, medical devices and military systems. When bugs appear in these systems the result can be catastrophic. Software also appears in *general-purpose* systems like cell-phones, smart-devices, and web infrastructure like email, banking and e-commerce. Bugs in general-purpose software systems are also costly: malware on cell-phones and websites compromise user privacy, and bugs in web-infrastructure lead to cyber-attacks, data corruption and service failures, costing billions of dollars annually [Tassey, 2002, Zhivich and Cunningham, 2009].

Achieving Software Reliability To achieve *high assurance* for software, we must establish the absence of entire classes of bugs and/or conformance with specific behavior. For example, a high-assurance medical device should not only be immune to a well-specified class of security exploits, it should also guaranteed to perform its intended medical function for the patient. Establishing high assurance is challenging, and current techniques are either unable to achieve it or too costly to adopt for many important applications.

What we *do* know is that *testing* software is not enough on its own to achieve high assurance. Most software systems have an infinite number of possible input/output behaviors, and testing is restricted to exploring only a finite subset of such behaviors. To achieve high assurance, one must use *verification* tools, which are able to reason symbolically about the infinite behavior of software.

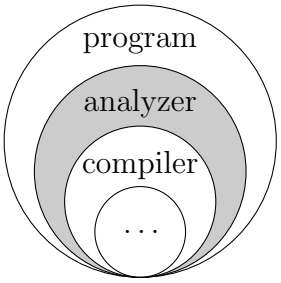
Program Analyzers This thesis considers *program analyzers*, which are tools for automating large portions of the verification proofs required to establish high assurance in software. Our results address the limitations of current approaches to building program analyzers, and contribute towards increased adoptions of tools which achieve high assurance in settings where software reliability remains an expensive and unsolved problem.

The Importance of Reusable Tools In order to have a positive impact on the way we produce software, program analyzers must not only be usable, they must be *re*usable. New programming languages are invented every year, for which

we lack tools like program analyzers. Likewise, new analysis techniques are also invented every year, for which implementations only exist for our oldest, most decrepit programming languages. What is missing is program analysis machinery which supports reuse across new programming languages, emerging software domains, and changing software correctness criteria.

To achieve *reuse* in program analyzer implementations, support for programming language features (*e.g.*, while loops) must be isolated from analysis properties (*e.g.*, buffer overflows). Techniques exist for isolating simple properties like arithmetic relationships (*e.g.*, $x < y$), but not for sophisticated properties like those used for security (*e.g.*, passwords are not leaked). Even in cases where *implementation* fragments can be reused, it is not possible to reuse the *proof* fragments used to establish the correctness of the resulting analyzer.

The Importance of Verified Tools Software is created through a complex pipeline: a program is written in a programming language, translated to machine code by a compiler, loaded by an operating system, and executed with hardware. To gain any amount of trust in the result, each component of the pipeline should be trustworthy. For this reason, it is just important to achieve *verified software tools* as it is to achieve the end goal of *verified software*, for the latter is not achievable without the former. Integrating program analyzers into this pipeline (see figure) comes with challenges similar to designing a compiler: implementations are often not reusable, and the



Software Pipeline

correctness of the tool must be established to achieve high assurance in the resulting software. This means analyzers must often be written from scratch to support new programming languages, and these new analyzers must then be verified if their results are to be trusted.

Mechanized Verification To achieve *high assurance* in program analyzer implementations, the gold standard is *mechanized verification* using automated theorem provers or semi-automated proof assistants. Consider for instance the compiler phase of the pipeline: a recent study showed that each of the 11 industry-strength C compilers examined had correctness bugs [Yang et al., 2011]. One of these compilers was CompCert, a mechanically verified C compiler [Leroy, 2009], however the only bugs present were in the unverified front-end. Program analyzers are similarly complex components of the software pipeline, and—like compilers—mechanized verification is the only technique known which can guarantee the absence of bugs in an implementation. For mechanization, the state of the art is to use proof assistants based on dependent type theory, which support extraction of certified algorithms.

Contributions This thesis improves the state of the art for both lightweight and heavyweight verification. One goal is for practitioners to eventually use *at least* lightweight verification for every piece of software—there is no reason not to. Another goal is to achieve heavyweight verification for mission critical software in settings which weren't possible or feasible before.

This thesis addresses reuse and high assurance—and their combination—for

program analyzer implementations, and our overarching insight is to *tightly couple the implementation of a program analyzer with its proof of correctness*. By tightly coupling implementation with proof, we design building blocks for constructing reliable analyzers from reusable components, and identify ways to reduce the proof effort required to mechanically verify analyzers.

High Assurance for Analyzer Implementations In this thesis we develop a new mathematical framework called *Constructive Galois Connections* (CGCs) [Darais and Van Horn, 2016] to mechanically verify a large class high-assurance program analyzers which previous approaches were unable to verify. These analyzers are called *correct-by-construction* because the implementation and proof of correctness for the analyzer are tightly coupled throughout their definition. Correct-by-construction analyzers are advantageous for mechanized verification because there is only one artifact to verify (the coupled implementation/proof), rather than two artifacts (the uncoupled implementation and proof), effectively reducing the proof burden by half. Constructing analyzers in this way also has the benefit of catching implementation bugs early, because incorrect implementation are not even possible to define. Central to these correct-by-construction program analyzers is a mathematical theory of sound approximation called *abstract interpretation* [Cousot and Cousot, 1977], however this theory is fundamentally limited in ways that prevent mechanized verification.

To design CGCs we addressed the limitations of abstract interpretation by re-instantiating the more general mathematical theory of *adjunctions*, of which abstract

interpretation is one instance. CGCs are an alternative instantiation of adjunctions which supports defining correct-by-construction program analyzers, but doesn't suffer from the same limitations to mechanized verification. One result of CGCs is the first mathematical foundation for program analyzers which simultaneously supports correct-by-construction design and mechanized verification. Other results from CGCs are case studies which construct the first mechanically verified and correct-by-construction program analyzer, as well as other mechanically verified applications which benefit from using abstract interpretation.

Reuse for Analyzer Implementations In this thesis we develop a program analysis framework called *Galois Transformers* (GTs) [Darais et al., 2015] to build reliable program analyzers from reusable components. The central principle of GTs is to unify the mathematical design of the analyzer with its implementation using *executable* state transition systems [Van Horn and Might, 2010]. However, state transition systems are not reusable across programming languages or for obtaining variations in analyzer precision. GTs solve half of the reuse problem for program analyzers—reuse of analyzer precision—by enriching the general structure of state transitions, and by implementing analysis machinery within this structure. As a result, GTs support implementing one important aspect of precision called *path and flow sensitivity* in a library. This library can be reused to construct new analyzers which feature path and flow sensitive precision for arbitrary programming languages, and without re-implementing complex analysis machinery.

Building on GTs, we develop a program analysis framework called *Abstracting*

Definitional Interpreters (ADI) [Darais et al., 2017], also for the purpose of building reliable analyzers from reusable components. ADI solves the other half of the reuse problem for program analyzers—reuse of programming language features—by adopting programming language interpreters in place of state transition systems as the unified platform for designing and implementing analyzers. As a result, ADI supports implementing analysis machinery for individual programming language features in a library, as well as a second important variation in analysis precision called *pushdown precision*. In combination with the results of GTs, ADI supports rapidly prototyping reliable program analyzers using reusable components: first for the features of the programming language being analyzed, and second for obtaining variations in analysis precision required by the application domain.

1.1 Outline

The remainder of this thesis is structured as follows: Chapter 2 presents necessary technical background. Chapter 3 presents an overview of the technical problems, main ideas, and evaluation methods presented in the remainder of the thesis. Chapters 4, 5 and 6 present the three main results of this thesis: Constructive Galois Connections, Galois Transformers and Abstracting Definitional Interpreters respectively. Chapter 7 concludes, and Chapter A provides supplementary proofs for Galois Transformer theorems from Chapter 5.

Chapter 2: Technical Background

2.1 Abstract Interpretation

Abstract Interpretation is a foundational framework for designing and implementing program analyzers and type systems—as well as a plethora of other useful programming language tools [Cousot, 2008]—invented and developed by Cousot and Cousot [1999, 1976, 1977, 1979, 1992, 1994, 2014].

At a high level, the goal of abstract interpretation is to make precise what it means for some collection of objects to be an "abstraction" of another, and what it means for operations over abstract objects to be "representative" of operations over the objects which they abstract. This concept of abstraction is made precise as a particular mathematical relationship between sets.

For example, any classification hierarchy can be seen as an abstraction, such as the class of "fruit," for which both "apples" and "mangos" are represented. There are operations which can performed on fruit, such as juicing—which turns "apples" into "apple juice"—blending—which turns "apples" into "an apple smoothie"—and slicing plus dehydrating—which turns "apples" into "apple chips." Likewise, these operations can be performed on "mangos," with similar results.

In the framework of abstract interpretation, the notion of an "abstract oper-

ation" is made precise such that one can specify each of juicing, blending, slicing and dehydrating at the abstract level of "fruit." These abstract operations for creating "fruit juice," "fruit smoothies" and "fruit chips" can then be shown to be compatible with all of the representative elements of "fruit," that is "apples," "mangos," "pineapples" etc.

We apply abstract interpretation in this thesis not for the purposes of describing fruit operations, but for describing *program analyzers* and their *correctness criteria*. The "apples" and "mangos" in this setting are computer programs—like the ones that implement Google's search algorithm, or instruct the camera on your mobile phone to take a photo and store the contents in memory. The "fruit" in this setting are abstract classifications of these programs such as "safe," "secure" or "efficient." By making a formal connection between a particular program (the "apple") and a property of interest like safety (the "fruit"), we design algorithms which automatically check whether or not this property holds—and justify their correctness—all within the guiding mathematical framework of abstract interpretation.

2.1.1 Galois Connection Mappings

Central to the framework of abstract interpretation is a mathematical structure called a *Galois connection*, consisting of an *abstraction mapping* α which maps from a *concrete domain* C to an *abstract domain* A, and a *concretization mapping* γ which maps in the reverse direction.[1] In general, the concrete and abstract domains are

[1] According to Cousot, the abstraction and concretizatio mappings are notated α and γ because they appear as the first and third letters of the greek alphabet, which mirror "a" and "c", the first letters for each mapping, and the first and third letters of the english alphabet.

partially ordered sets, and the mappings are required to be *monotonic*, which we notate with an upward slanted arrow.

$$\begin{array}{rlrl} \textit{(concrete domain)} & C \ : \ \textit{poset} & \textit{(abstraction mapping)} & \alpha \ : \ C \nearrow A \\ \textit{(abstract domain)} & A \ : \ \textit{poset} & \textit{(concretization mapping)} & \gamma \ : \ A \nearrow C \end{array}$$

EXAMPLE Consider a very simple abstraction: the latin alphabet ($\mathcal{L}$) which includes both lowercase and uppercase letters, and the *logical* latin alphabet ($\widehat{\mathcal{L}}$) which unifies the letters `a` and `A` as the same "logical" letter.

$$\begin{array}{rl} \textit{(latin characters)} & \mathcal{L} = \{\mathtt{a}, \mathtt{A}, \ldots, \mathtt{z}, \mathtt{Z}\} \\ \textit{(logical characters)} & \widehat{\mathcal{L}} = \{\mathtt{A}, \ldots, \mathtt{Z}\} \end{array}$$

We represent "logical" letters as the uppercase form. In order to map between each set, we lift them to *powersets*. This means elements of the *concrete domain* are sets of latin characters (*e.g.*, $\{\mathtt{x}, \mathtt{Y}, \mathtt{z}\}$), and elements of the *abstract domain* are sets of logical characters (*e.g.*, $\{\mathtt{X}, \mathtt{Y}, \mathtt{Z}\}$).

$$\begin{array}{rl} \textit{(concrete domain)} & C \coloneqq \wp(\mathcal{L}) \\ \textit{(abstract domain)} & A \coloneqq \wp(\widehat{\mathcal{L}}) \end{array}$$

The abstraction function (α) maps a set of latin characters (*e.g.*, $\{\mathtt{x}, \mathtt{y}, \mathtt{Y}, \mathtt{Z}\}$) to the set of logical characters in that set (*e.g.*, $\{\mathtt{X}, \mathtt{Y}, \mathtt{Z}\}$). The concretization function is not an inverse mapping, rather it maps set of logical characters to the *smallest* set of latin characters which contains *every* set that abstracts to the original logical set of characters. For example, the concretization of $\{\mathtt{X}, \mathtt{Y}, \mathtt{Z}\}$ is $\{\mathtt{x}, \mathtt{X}, \mathtt{y}, \mathtt{Y}, \mathtt{z}, \mathtt{Z}\}$, because it is the smallest set that contains $\{\mathtt{x}, \mathtt{y}, \mathtt{z}\}$, $\{\mathtt{X}, \mathtt{y}, \mathtt{z}\}$, $\{\mathtt{x}, \mathtt{X}, \mathtt{y}, \mathtt{z}\}$, $\{\mathtt{x}, \mathtt{Y}, \mathtt{z}\}$, etc., each of which abstract to $\{\mathtt{X}, \mathtt{Y}, \mathtt{Z}\}$. For this example, we notate the pointwise abstraction of a single latin character η, and the pointwise concretization of a single logical

character μ.

$$\alpha : \wp(\mathcal{L}) \nearrow \wp(\widehat{\mathcal{L}}) \qquad \eta : \mathcal{L} \to \widehat{\mathcal{L}} \qquad \alpha(X) \coloneqq \{\eta(x) \mid x \in X\} \qquad \begin{array}{l} \eta(\mathtt{a}) \coloneqq \mathtt{A} \\ \quad \cdots \\ \eta(\mathtt{z}) \coloneqq \mathtt{Z} \end{array}$$

$$\gamma : \wp(\widehat{\mathcal{L}}) \nearrow \wp(\mathcal{L}) \qquad \mu : \widehat{\mathcal{L}} \to \wp(\mathcal{L}) \qquad \gamma(Y) \coloneqq \bigcup_{y \in Y} \mu(y)$$

$$\begin{array}{ll} \eta(\mathtt{A}) \coloneqq \mathtt{A} & \mu(\mathtt{A}) \coloneqq \{\mathtt{a}, \mathtt{A}\} \\ \quad \cdots & \quad \cdots \\ \eta(\mathtt{Z}) \coloneqq \mathtt{Z} & \mu(\mathtt{Z}) \coloneqq \{\mathtt{z}, \mathtt{Z}\} \end{array}$$

2.1.2 Galois Connection Laws

In addition to mapping between concrete and abstract domains, a Galois connection $\langle \alpha, \gamma \rangle$ must obey the following laws:

$$X \sqsubseteq \gamma(\alpha(X)) \qquad \text{(GC-Extensive)}$$

$$\alpha(\gamma(Y)) \sqsubseteq Y \qquad \text{(GC-Reductive)}$$

or equivalently, the following correspondence:

$$X \sqsubseteq \gamma(Y) \iff \alpha(X) \sqsubseteq Y \qquad \text{(GC-Corr)}$$

Example Continuing the previous example: consider the collection of characters $c \coloneqq \{\mathtt{X}, \mathtt{y}, \mathtt{Y}, \mathtt{z}\}$. We can describe which logical characters are contained in c as $\alpha(c) = \{\mathtt{X}, \mathtt{Y}, \mathtt{Z}\}$. We can also describe which literal characters are represented by this set of logical characters as $\gamma(\alpha(c)) = \{\mathtt{x}, \mathtt{X}, \mathtt{y}, \mathtt{Y}, \mathtt{z}, \mathtt{Z}\}$. However, repeatedly applying

α and γ eventually converges:

$$\begin{aligned} c &= \{X, y, Y, z\} \\ \alpha(c) &= \{X, Y, Z\} \\ \gamma(\alpha(c)) &= \{x, X, y, Y, z, Z\} \\ \alpha(\gamma(\alpha(c))) &= \{X, Y, Z\} && = \alpha(c) \\ \gamma(\alpha(\gamma(\alpha(c)))) &= \{x, X, y, Y, z, Z\} && = \gamma(\alpha(c)) \end{aligned}$$

What (GC-Extensive) ensures in our example is that $\gamma(\alpha(c)) \supseteq c$, or that "the abstraction for c ($\alpha(c)$) includes c in its representation ($\gamma(\alpha(c))$)." The second law (GC-Reductive) ensures that $\alpha(\gamma(\alpha(c))) \subseteq \alpha(c)$, or that "the abstraction for c ($\alpha(c)$) is no smaller than the abstraction of its representation ($\alpha(\gamma(\alpha(c)))$)." Repeated applications of α and γ (in either direction) will necessarily converge after one iteration, which is referred to as *idempotenecy*:

$$\gamma(\alpha(\gamma(\alpha(X)))) = \gamma(\alpha(X))$$

$$\alpha(\gamma(\alpha(\gamma(Y)))) = \alpha(\gamma(Y))$$

(Idempotency follows as a consequence of (GC-Extensive), (GC-Reductive), and partial order antisymmetry.)

It is often the case (as in our example) that a stronger form of (GC-Reductive) holds, that is with an equality rather than partial ordering:

$$\alpha(\gamma(Y)) = Y \qquad \text{(GC-Red-Strict)}$$

at which point the Galois connection is called a *Galois insertion*, or *Galois surjection*.

2.1.3 Abstract Interpreters

The structure of a Galois connection $\langle \alpha, \gamma \rangle$ determines both the meaning of *soundness* and *optimality* for an *abstract operation* ($\widehat{f}$)—which maps between elements of the abstract domain ($A \nearrow A$)—w.r.t. a *concrete operation* (f)—which maps between elements of the concrete domain ($C \nearrow C$).

$$\begin{aligned} \textit{concrete operation:} \quad & f \ : \ C \nearrow C \\ \textit{abstract operation:} \quad & \widehat{f} \ : \ A \nearrow A \end{aligned}$$

Soundness or optimality is then demonstrated by relating the abstract operation ($\widehat{f}$) to an optimal specification induced from the concrete operation ($\alpha \circ f \circ \gamma$), either using a partial order to establish soundness, or equality to establish optimality.

$$\alpha \circ f \circ \gamma \sqsubseteq \widehat{f} \qquad \text{(GC-Sound)}$$

$$\alpha \circ f \circ \gamma = \widehat{f} \qquad \text{(GC-Optimal)}$$

Example Continuing the running example: consider the concrete operation of concatenating two latin characters together to form a string, notated $c_1 +\!\!+ c_2$, *e.g.*, $\texttt{x} +\!\!+ \texttt{y} = \texttt{xy}$. We lift this operation to operate over powersets in order to express concatenation over *properties* of characters ($\wp(\mathcal{L})$), which is also the concrete domain C. This is often called the *collecting* semantics, because it supports expressing properties of interest over inputs and outputs to the operation, encoded as powersets.

$$\widetilde{+\!\!+} \ : \ \wp(\mathcal{L}) \times \wp(\mathcal{L}) \nearrow \wp(\mathcal{L} \times \mathcal{L}) \qquad X_1 \ \widetilde{+\!\!+} \ X_2 \ \coloneqq \ \{c_1 +\!\!+ c_2 \mid c_1 \in X_1 \wedge c_2 \in X_2\}$$

An abstraction of this operation $Y_1 \mathbin{\widehat{+\!\!+}} Y_2$ is considered *sound* if the abstract concatenation of two sets of logical characters Y_1 and Y_2 *contains* all of the concatenations of sets of concrete characters $\gamma(Y_1)$ and $\gamma(Y_2)$, and *optimal* if the abstract concatenation of two sets of logical characters Y_1 and Y_1 is *equal to* all of the concrete concatenations. These are exactly the notions generated by the induced specifications (GC-Sound) and (GC-Optimal).

It turns out that for this example, abstract concatenation is identical to concrete concatenation, that is:

$$\mathbin{\widehat{+\!\!+}} : \wp(\widehat{\mathcal{L}}) \times \wp(\widehat{\mathcal{L}}) \nearrow \wp(\widehat{\mathcal{L}} \times \widehat{\mathcal{L}}) \qquad Y_1 \mathbin{\widehat{+\!\!+}} Y_2 \coloneqq \{d_1 +\!\!+ d_2 \mid d_1 \in Y_1 \wedge d_2 \in Y_2\}$$

The statement that abstract concatenation ($\widehat{+\!\!+}$) is a sound approximation of the concrete concatenation ($\widetilde{+\!\!+}$) is induced by the Galois connection defined previously:

$$\alpha(\gamma(Y_1) \mathbin{\widetilde{+\!\!+}} (Y_2)) \sqsubseteq Y_1 \mathbin{\widehat{+\!\!+}} Y_2$$

although its proof is nontrivial, and likewise for optimality:

$$\alpha(\gamma(Y_1) \mathbin{\widetilde{+\!\!+}} (Y_2)) = Y_1 \mathbin{\widehat{+\!\!+}} Y_2$$

2.1.4 Calculational Abstract Interpretation

Rather than postulate the definition of an abstract operation ($\widehat{f}$) and verify its soundness or optimality (*via* (GC-Sound) or (GC-Optimal)), one can instead derive a sound or optimal implementation directly from the induced specification. The chain of reasoning begins on the left-hand side with the induced optimal specification—

which is often not directly implementable as an algorithm—and proceeds through directed rewrites of the specification. At some point, the current state of reasoning is observed to have algorithmic content, or can be easily translated into an algorithm, and is declared to be the implementation of the abstract operator:

$$\alpha(f(\gamma(Y))) \ldots \sqsubseteq \ldots \sqsubseteq \ldots \triangleq \widehat{f}$$

If directed reasoning was used, as shown in the above mock-derivation, then the result is guaranteed to be sound *by construction*. If purely equational reasoning was used—so equalities ($=$) for each step rather than partial orders ($\sqsubseteq$)—then the result is guaranteed to be optimal *by construction* as well.

EXAMPLE Continuing the running example: we will now derive a sound and optimal abstract concatenation operator using calculational abstract interpretation.

First, the concrete collecting operation ($\widetilde{+\!\!+}$) is lifted to a specification for an abstract operation through composition with abstraction and concretization mappings. We demonstrate the calculation using a specific instantiation of parameters Y_1 and Y_2, and later generalize the result. For now, consider $Y_1 = \{\mathtt{X}, \mathtt{Y}\}$ and $Y_2 = \{\mathtt{Z}\}$:

$$\alpha(\gamma(\{\mathtt{X}, \mathtt{Y}\}) \mathbin{\widetilde{+\!\!+}} \gamma(\{\mathtt{Z}\})) = \ldots$$

The first step in the calculation is to apply the concretization mapping (γ):

$$\ldots = \alpha(\{\mathtt{x}, \mathtt{X}, \mathtt{y}, \mathtt{Y}\} \mathbin{\widetilde{+\!\!+}} \{\mathtt{z}, \mathtt{Z}\}) = \ldots$$

The next step is to apply the collecting concatenation operation ($\widetilde{+\!\!+}$):

$$\ldots = \alpha(\{\texttt{xz}, \texttt{yz}, \texttt{Xz}, \texttt{Yz}, \texttt{xZ}, \texttt{yZ}, \texttt{XZ}, \texttt{YZ}\}) = \ldots$$

The final step is to apply the abstraction mapping (α), after which we declare the result the implementation of abstract concatenation:

$$\ldots = \{\texttt{XZ}, \texttt{YZ}\} \triangleq \{\texttt{X}, \texttt{Y}\} \mathbin{\widehat{+\!\!+}} \{\texttt{Z}\}$$

To generalize over arbitrary inputs Y_1 and Y_2, the derivation is carried out symbolically. The first step applies concretization, effectively containing the union of Y_1 interpreted as both uppercase and lowercase, and likewise for Y_2. The next step applies the collecting concatenation of these sets, which interleaves every possible combination of uppercase and lowercase. The final step applies abstraction, which eliminates redundant occurrences of uppercase and lowercase in the concrete set:

$$\begin{aligned}
&\alpha(\gamma(Y_1) \mathbin{\widetilde{+\!\!+}} \gamma(Y_2)) \\
&= \quad \wr \quad \text{applying } \gamma \quad \int \\
&\alpha((\mathit{upper}(Y_1) \cup \mathit{lower}(Y_1)) \mathbin{\widetilde{+\!\!+}} (\mathit{upper}(Y_2) \cup \mathit{lower}(Y_2))) \\
&= \quad \wr \quad \text{applying } \widetilde{+\!\!+} \quad \int \\
&\alpha\left(\bigcup \left\{ \begin{array}{l} \{x_1x_2 \mid x_1 \in \mathit{upper}(Y_1) \wedge x_2 \in \mathit{upper}(Y_2)\} \\ \{x_1x_2 \mid x_1 \in \mathit{lower}(Y_1) \wedge x_2 \in \mathit{upper}(Y_2)\} \\ \{x_1x_2 \mid x_1 \in \mathit{upper}(Y_1) \wedge x_2 \in \mathit{lower}(Y_2)\} \\ \{x_1x_2 \mid x_1 \in \mathit{lower}(Y_1) \wedge x_2 \in \mathit{lower}(Y_2)\} \end{array} \right.\right) \\
&= \quad \wr \quad \text{applying } \alpha \quad \int \\
&\{y_1y_2 \mid y_1 \in Y_1 \wedge y_2 \in Y_2\} \\
&\triangleq \quad \wr \quad \text{by defining } Y_1 \mathbin{\widehat{+\!\!+}} Y_2 \coloneqq \{y_1y_2 \mid y_1 \in Y_1 \wedge y_2 \in Y_2\} \quad \int \\
&Y_1 \mathbin{\widehat{+\!\!+}} Y_2 \quad \blacksquare
\end{aligned}$$

2.1.5 Conclusion

In this section we reviewed the essential structure of calculational abstract interpretation—both through its general definition, and through a simple running example based on an abstraction for latin characters which ignores whether or not a character is uppercase or lowercase. The capstone of the exercise was an implementation for an abstract concatenation operator, which was derived by calculus, and is therefore both sound and optimal by construction.

The remainder of this thesis will make heavy use of abstract interpretation as a technique for justifying the soundness and optimality of program analyzers. One of the contributions in this thesis is an alternative setup for abstract interpretation called *constructive Galois connections* which allows deriving algorithms which not only sound or optimal, but *computable* by construction as well.

2.2 Abstracting Abstract Machines

Abstracting Abstract Machines (AAM) is a technique for systematically deriving program analyzers directly from a description of that programming language, invented by Van Horn and Might [2010, 2012].

At a high level, the goal of AAM is to make designing program analyzers easier, and in particular for new and feature-rich programming languages. A program analyzer is essentially an algorithm which attempts to predict the behavior of individual programs, typically by classifying programs as either "definitely good" or "possibly bad." To justify the correctness of the prediction, an exercise must be

performed which examines the semantics of the programming language—*i.e.* a formal description of what individual programs "mean"—and the content of the program analysis algorithm. Given these two artifacts, the exercise is to establish that every result computed by the algorithm offers a reliable prediction of the behavior of the program being analyzed.

The core approach of AAM is:

1. To describe the semantics of the programming language using *small-step operational semantics* [Felleisen and Hieb, 1992, Plotkin, 1981]; and

2. A technique for systematically abstracting a *concrete* small-step semantics into an *abstract* small-step semantics

Because the technique is systematic, it leaves little room for error (a good thing) or complex analysis techniques (a limitation). Therefore, to recover complex analysis techniques, the essence of the technique must be embedding in the *concrete* version of the semantics, such that it is present in the program analyzer after systematic abstraction.

Although the abstraction process is mostly systematic, there is one parameter exposed after abstraction which has a large determining factor on the resulting program analysis: *abstract allocation*. In order to execute the program analysis, one must define an allocation strategy, and different strategies give rise to a wide range of possible program analysis techniques, with varying precision and performance tradeoffs [Gilray et al., 2016a].

2.2.1 Small-step Semantics

Central to the Abstracting Abstract Machines (AAM) technique [Van Horn and Might, 2010, 2012] is the setting of *small-step operational semantics* [Felleisen and Hieb, 1992, Felleisen et al., 1987, Plotkin, 1981]. A small-step operational semantics for a programming language is a mathematical *relation* between purely *syntactic* terms, which describes a relatively *small* unit of computation. The reflexive-transitive-closure of this relation is then taken to describe *evaluation* for the programming language, which fully reduces a program text to the output it computes.

One distinguishing feature between approaches to small-step semantics is the treatment of the *context* of sub-computations. The approach we take is to treat contexts as an explicit object in the reduction system, *à la* Felleisen and Friedman's CEK machine [1987].

EXAMPLE Consider a very simple programming language for adding natural numbers, *e.g.*, 5, $\texttt{PLUS}(1,2)$ and $\texttt{PLUS}(\texttt{PLUS}(1,2),\texttt{PLUS}(3,4))$ are all valid programs. The syntax for this language is described in BNF [Backus, 1959]:

$$\begin{aligned} n \in \quad \mathbb{N} &::= \{0,1,3,4,\ldots\} \\ e \in \mathit{exp} &::= n \mid \texttt{PLUS}(e,e) \end{aligned}$$

To represent contexts explicitly in the semantics, we define a language for evaluation contexts, and define configurations as a pairing of an expression to evaluate, and the

context for the evaluation:

$$\kappa \in \mathit{context} ::= \texttt{PLUS}(\square, e) :: \kappa \mid \texttt{PLUS}(e, \square) :: \kappa \mid \texttt{HALT}$$
$$\varsigma \in \mathit{config} := \mathit{exp} \times \mathit{context}$$

The small-step semantics for these expressions is then expressed as a set of relational rules, describing in which configurations a step of computation occurs:

$$\textit{(Small-step Evaluation)} \quad \boxed{\varsigma \rightsquigarrow \varsigma}$$

$$\textit{(Plus)} \quad \langle \texttt{PLUS}(n_1, n_2), \kappa \rangle \rightsquigarrow \langle n_1 + n_2, \kappa \rangle$$
$$\textit{(PPushL)} \quad \langle \texttt{PLUS}(e_1, e_2), \kappa \rangle \rightsquigarrow \langle e_1, \texttt{PLUS}(\square, e_2) :: \kappa \rangle$$
$$\textit{(PPushR)} \quad \langle \texttt{PLUS}(e_1, e_2), \kappa \rangle \rightsquigarrow \langle e_2, \texttt{PLUS}(e_1, \square) :: \kappa \rangle$$
$$\textit{(PPopL)} \quad \langle n, \texttt{PLUS}(\square, e_2) :: \kappa \rangle \rightsquigarrow \langle \texttt{PLUS}(n, e_2), \kappa \rangle$$
$$\textit{(PPopR)} \quad \langle n, \texttt{PLUS}(e_1, \square) :: \kappa \rangle \rightsquigarrow \langle \texttt{PLUS}(e_1, n), \kappa \rangle$$

This relation is nondeterministic; for example, both

$$\langle \texttt{PLUS}(\texttt{PLUS}(1,2), \texttt{PLUS}(3,4)), \texttt{HALT} \rangle \rightsquigarrow \langle \texttt{PLUS}(1,2), \texttt{PLUS}(\square, \texttt{PLUS}(3,4)) :: \texttt{HALT} \rangle$$

$$\langle \texttt{PLUS}(\texttt{PLUS}(1,2), \texttt{PLUS}(3,4)), \texttt{HALT} \rangle \rightsquigarrow \langle \texttt{PLUS}(3,4), \texttt{PLUS}(\texttt{PLUS}(1,2), \square) :: \texttt{HALT} \rangle$$

are described by the relation. The reflexive-transitive-closure is often notated $\rightsquigarrow^*$, and for our example relates the example program to its evaluation result of 10:

$$\langle \texttt{PLUS}(\texttt{PLUS}(1,2), \texttt{PLUS}(3,4)), \texttt{HALT} \rangle \rightsquigarrow^* \langle 10, \texttt{HALT} \rangle$$

2.2.2 Adding Higher-order Functions

To transition our concrete semantics to an abstract semantics, the primary goal is to achieve a *finite state space* for the domain of the relation. The reason for this is so that all the behavior of a program can be explored in finite time, which constitutes a decidable program analysis algorithm. This becomes challenging for

inductively defined components of the domain, and particularly challenging for mutually inductively defined components. The AAM approach to this problem is to introduce an explicit level of indirection between recursive occurrences of a structure, and to apply an allocation mechanism for referencing child-structures from parent-structures.

Example Continuing the running example: currently the only source of non-finiteness in the state space for the relation ($\wp(\mathit{exp} \times \mathit{exp})$) is the set of natural numbers ($\mathbb{N}$). Thus, the goal of abstracting the current semantics poses no great challenge, and the AAM technique doesn't necessarily apply. Let's extend the language with *higher-order functions*, *i.e lambda*-terms, to see the AAM technique at work. First, we add variables (x), anonymous functions ($\lambda x.\, e$), and function application ($e(e)$) as syntactic terms to the expression language:

$$\begin{aligned} n \in \quad \mathbb{N} &::= \{0, 1, 3, 4, \ldots\} \\ x \in \mathit{var} &::= \{\mathtt{x}, \mathtt{y}, \ldots\} \\ e \in \mathit{exp} &::= n \mid \mathtt{PLUS}(e, e) \mid x \mid \lambda x.\, e \mid e(e) \end{aligned}$$

Next, we add *environments* to the domain of configurations, closures ($\langle \lambda x.\, e, \rho \rangle$) to the domain of values and control expressions, and extend contexts to carry the

environment under which that computation was initiatied:

$$
\begin{array}{rrcl}
v \in & \mathit{val} & ::= & \mathbb{N} \cup \{\langle \lambda x.\ e, \rho \rangle\} \\
\rho \in & \mathit{env} & ::= & \mathit{var} \rightharpoonup \mathit{val} \\
c \in & \mathit{control} & ::= & v \mid n \mid \texttt{PLUS}(c, c) \mid x \mid \lambda x.\ e \mid c(c) \\
\kappa \in & \mathit{context} & ::= & \langle \texttt{PLUS}(\square, c), \rho \rangle :: \kappa \mid \langle \texttt{PLUS}(c, \square), \rho \rangle :: \kappa \\
 & & \mid & \langle \square(c), \rho \rangle :: \kappa \mid \langle c(\square), \rho \rangle :: \kappa \mid \texttt{HALT} \\
\varsigma \in & \mathit{config} & ::= & \mathit{control} \times \mathit{env} \times \mathit{context}
\end{array}
$$

and we add the following rules to the small-step semantic relation:

$$
\begin{array}{rc}
 & \textit{(Small-step Evaluation)} \quad \boxed{\varsigma \rightsquigarrow \varsigma} \\
\textit{(Var)} & \langle x, \rho, \kappa \rangle \rightsquigarrow \langle \rho(x), \rho, \kappa \rangle \\
\textit{(Lam)} & \langle \lambda x.\ e, \rho, \kappa \rangle \rightsquigarrow \langle \langle \lambda x.\ e, \rho \rangle, \rho, \kappa \rangle \\
\textit{(Apply)} & \langle \langle \lambda x.\ e, \rho' \rangle (v), \rho, \kappa \rangle \rightsquigarrow \langle e, \rho'[x \mapsto v], \kappa \rangle \\
\textit{(APushL)} & \langle c_1(c_2), \rho, \kappa \rangle \rightsquigarrow \langle c_1, \rho, \langle \square(c_2), \rho \rangle :: \kappa \rangle \\
\textit{(APushR)} & \langle c_1(c_2), \rho, \kappa \rangle \rightsquigarrow \langle c_2, \rho, \langle c_1(\square), \rho \rangle :: \kappa \rangle \\
\textit{(APopL)} & \langle v, \rho, \langle \square(e), \rho' \rangle :: \kappa \rangle \rightsquigarrow \langle v(e), \rho', \kappa \rangle \\
\textit{(APopR)} & \langle v, \rho, \langle e(\square), \rho' \rangle :: \kappa \rangle \rightsquigarrow \langle e(v), \rho', \kappa \rangle
\end{array}
$$

2.2.3 Adding Indirection through a Store

Now our language supports higher-order-functions, but it has become much harder to abstract and finitize the state space. The key challenge is how to abstract contexts, which are defined recursively, and how to abstract *both* values and environments, which are defined mutually recursively. The AAM solution to abstraction in this setting is to introduce an explicit level of indirection through a store for recursively defined constructs.

EXAMPLE To continue the running example: we modify the language with a level of indirection between linked contexts, and between values and environments:

$$
\begin{array}{rrcl}
v \in & \mathit{val} & \coloneqq & \mathbb{N} \cup \{\langle \lambda x.\, e, \rho \rangle\} \\
\ell \in & \mathit{addr} & \coloneqq & \textit{(parameter)} \\
\rho \in & \mathit{env} & \coloneqq & \mathit{var} \rightharpoonup \mathit{addr} \\
\sigma \in & \mathit{store} & \coloneqq & \mathit{addr} \rightharpoonup \mathit{val} \cup \mathit{context} \\
c \in & \mathit{control} & ::= & v \mid n \mid \texttt{PLUS}(c, c) \mid x \mid \lambda x.\, e \mid c(c) \\
\kappa \in & \mathit{context} & \coloneqq & \langle \texttt{PLUS}(\square, c), \rho \rangle :: \ell \mid \langle \texttt{PLUS}(c, \square), \rho \rangle :: \ell \\
 & & \mid & \langle \square(c), \rho \rangle :: \ell \mid \langle c(\square), \rho \rangle :: \ell \mid \texttt{HALT} \\
\varsigma \in & \mathit{config} & \coloneqq & \mathit{control} \times \mathit{env} \times \mathit{store} \times \mathit{context}
\end{array}
$$

which requires modifying each reduction rule, only four of which we show here:

$$
\begin{array}{rrcl}
 & & & \textit{(Small-step Evaluation)} \quad \boxed{\varsigma \rightsquigarrow \varsigma} \\
\textit{(Var)} & \langle x, \rho, \sigma, \kappa \rangle & \rightsquigarrow & \langle \sigma(\rho(x)), \rho, \sigma, \kappa \rangle \\
\textit{(Apply)} & \langle \langle \lambda x.\, e, \rho' \rangle (v), \rho, \sigma, \kappa \rangle & \rightsquigarrow & \langle e, \rho'[x \mapsto \ell], \sigma[\ell \mapsto v], \kappa \rangle \\
 & & \texttt{where } \ell & \coloneqq \textit{ fresh} \\
\textit{(APushL)} & \langle c_1(c_2), \rho, \sigma, \kappa \rangle & \rightsquigarrow & \langle c_1, \rho, \sigma[\ell \mapsto \kappa], \langle \square(c_2), \rho \rangle :: \ell \rangle \\
 & & \texttt{where } \ell & \coloneqq \textit{ fresh} \\
\textit{(APopL)} & \langle v, \rho, \sigma, \langle \square(e), \rho' \rangle :: \ell \rangle & \rightsquigarrow & \langle v(e), \rho', \sigma, \sigma(\ell) \rangle
\end{array}
$$

2.2.4 Abstraction

Once we have introduced indirection into the semantics to mediate between values and environments, we can then finitize the entire configuration space ς merely by picking finite abstractions for each of natural numbers ($\mathbb{N}$) and addresses (addr). Once finitizing abstractions are picked for numbers and addresses, then the AAM approach systematically constructs an abstract semantics, which are directly implementable as an executable program analyzer.

EXAMPLE To continue ther unning example, consider some abstraction for natural numbers $\widehat{\mathbb{N}}$ and abstraction for addresses $\widehat{addr}$, after which the state space becomes:

$$\begin{array}{rrcl}
v \in & \widehat{val} & \coloneqq & \widehat{\mathbb{N}} \cup \wp(\{\langle \lambda x.\, e, \rho \rangle\}) \\
\rho \in & \widehat{env} & \coloneqq & var \rightharpoonup \widehat{addr} \\
\sigma \in & \widehat{store} & \coloneqq & \widehat{addr} \to \widehat{val} \cup \wp(\widehat{context}) \\
c \in & \widehat{control} & ::= & v \mid n \mid \texttt{PLUS}(c, c) \mid x \mid \lambda x.\, e \mid c(c) \\
\kappa \in & \widehat{context} & \coloneqq & \langle \texttt{PLUS}(\square, c), \rho \rangle :: \ell \mid \langle \texttt{PLUS}(c, \square), \rho \rangle :: \ell \\
 & & \mid & \langle \square(c), \rho \rangle :: \ell \mid \langle c(\square), \rho \rangle :: \ell \mid \texttt{HALT} \\
\varsigma \in & \widehat{config} & \coloneqq & \widehat{control} \times \widehat{env} \times \widehat{store} \times \widehat{addr}
\end{array}$$

and the following six (only, for brevity) rules become:

(Small-step Evaluation) $\boxed{\varsigma \rightsquigarrow \varsigma}$

$$\begin{array}{lrcl}
\textit{(Plus)} & \langle \texttt{PLUS}(v_1, v_2), \rho, \sigma, \ell \rangle & \rightsquigarrow & \langle v_1 \mathbin{\widehat{+}} v_2, \rho, \sigma, \ell \rangle \\
\textit{(PPushL)} & \overbrace{\langle \texttt{PLUS}(c_1, c_2), \rho, \sigma, \ell \rangle}^{\varsigma} & \rightsquigarrow & \langle c_1, \rho, \sigma \sqcup [\ell' \mapsto \langle \texttt{PLUS}(\square, c_2), \rho \rangle :: \ell], \ell' \rangle \\
 & \texttt{where } \ell' & \coloneqq & alloc(\varsigma) \\
\textit{(PPopL)} & \langle v, \rho, \sigma, \ell \rangle & \rightsquigarrow & \langle \texttt{PLUS}(v, c_2), \rho', \sigma, \ell' \rangle \\
 & \texttt{where } \langle \texttt{PLUS}(\square, c_2), \rho' \rangle :: \ell' & \in & \sigma(\ell) \\
\textit{(Apply)} & \overbrace{\langle v_1(v_2), \rho, \sigma, \kappa \rangle}^{\varsigma} & \rightsquigarrow & \langle e, \rho'[x \mapsto \ell], \sigma \sqcup [\ell \mapsto v_2], \kappa \rangle \\
 & \texttt{where } \langle \lambda x.\, e, \rho' \rangle & \in & v_1 \\
 & \ell & \coloneqq & alloc(\langle \lambda x.\, e, \rho' \rangle, \varsigma)) \\
\textit{(APushL)} & \overbrace{\langle c_1(c_2), \rho, \sigma, \kappa \rangle}^{\varsigma} & \rightsquigarrow & \langle c_1, \rho, \sigma \sqcup [\ell' \mapsto \langle \square(c_2), \rho \rangle :: \ell], \ell' \rangle \\
 & \texttt{where } \ell' & \coloneqq & alloc(\varsigma) \\
\textit{(APopL)} & \langle v, \rho, \sigma, \ell \rangle & \rightsquigarrow & (v(c_2), \rho', \sigma, \ell' \rangle \\
 & \texttt{where } \langle \square(c_2), \rho' \rangle :: \ell' & \in & \sigma(\ell)
\end{array}$$

(The alteration of the rest of the rules is analogous.) Note in particular the transition to *multiple* elements found in the store on function calls and stack popping, and the *joining* of results in the store on function application and stack pushing, *e.g.*, when

the address already exists in the store from a separate binding instance.

2.2.5 Conclusion

In this section we reviewed small-step operational semantics and the AAM approach to systematic abstraction of a concrete to an abstract semantics. This was done through a running example which extended a simple arithmetic expression programming language into one which includes higher-order functions. Allocation was introduced to break the recursive structure of the state space, and finitization was achieved by finitizing the domains for natural numbers and addresses.

The remainder of this thesis will build heavily on the AAM approach to semantics abstraction. One of the contributions in this thesis is a technique called *Galois transformers* which introduces another parameter to the semantics alongside *alloc* for recovering variations in path and flow sensitivity. Another contribution in this thesis is a technique called *abstract definitional interpreters* which extends the AAM technique in full generality to the semantic setting of definitional interpreters (as opposed to operational small-step relations).

2.3 Mechanized Verification

Mechanized Verification is a technique for establishing the correctness of a piece of software with the highest possible confidence we know how to achieve. In mechanized verification, formal proofs are constructed to establish the correctness of a piece of software—down to the smallest detail—and computers are used to check the validity

of these proofs rather than an expert human. To gain the highest level of assurance, the software which *checks* the proofs must be small, simple, and easy to inspect by experts to establish *its* correctness.

The approach to mechanized verification for software considered in this thesis is that which uses (in spirit) *intuitionist type theory* (ITT) [Martin-Löf, 1975, 1984], as embodied in proof assistants like Coq [development team, 2004] and Agda [Norell, 2007], each of which are implementations based on the *calculus of inductive constructions* (CIC) [Coquand and Huet, 1988, 1985, Coquand and Paulin, 1990], a descendent of ITT. These proof assistants unify the language used to describe *programs*, the language used to describe *properties* of programs, and the language used to describe *proofs* of these properties. At the center of this unified framework is an intrinsic notion of *computation*, which each of programs, properties, and proofs must carry. As a consequence of this, classical reasoning principles—such as the Law of Excluded Middle (LEM)—are disallowed in the theory, because they do not carry computational content. (Although LEM can be added as an axiom without interfering with the logic's consistency, its use will interfere with the computational nature of the system.)

Because ITT terms carry computational content, they can be interpreted and run as programs, typically by extraction to a functional language like OCaml, Haskell, or Scheme. It is in this way that certified implementations of various algorithms (like program analyzers) are produced: by embedding the algorithm as well as its proof of correctness in a proof assistant, and then extracting a functional program from the algorithm description, which is run using a conventional functional programming

language compiler and runtime.

2.3.1 Equality

Central to any mechanized verification technique is a fundamental tradeoff between *automatic* proof construction (what you get for free), and *manual* proof construction (what you don't get for free). In ITT (and therefore CIC), this tradeoff is embodied in two distinct notions of equality: so-called *definitional* equality, and so-called *propositional* equality, respectively. Definitional equalities, notated with $\overset{\lightning}{=}$, are judgments which can be justified entirely through computation, *e.g.*, $1 + 2 \overset{\lightning}{=} 3$ or $[1, 2] +\!\!+ [3] \overset{\lightning}{=} [1, 2, 3]$. These equalities cannot be mentioned internally in the proof system, rather this equality is used as a congruence, that is the proof system is always considering the validity of judgemental equalities modulo definitional equality, *i.e.* modulo computation. Propositional equalities, notated with $\equiv$, are judgements which require a manually or semi-automatically constructed proof. When the judgment is an embedding of a definitional equality, *e.g.*, $1 + 2 \equiv 3$, then the trivial proof term `Refl` is sufficient evidence, because the system carries out the reasoning of equality modulo computation automatically. When the judgment is non-trivial, *e.g.*, $\forall x.\, x + 0 \equiv x$, then a proof must be supplied in the form of a witness term, which is also a program with computational content, due to the unification of these concepts.

EXAMPLE Consider the proof term of the right identity for addition in Agda:

$$\begin{aligned}
&\mathit{right\text{-}identity} \;:\; \forall\ x \to x + 0 \equiv x \\
&\mathit{right\text{-}identity}\ \texttt{Zero} = \texttt{Refl} \\
&\mathit{right\text{-}identity}\ (\texttt{Succ}\ x')\ \texttt{rewrite}\ \mathit{right\text{-}identity}\ x' = \texttt{Refl}
\end{aligned}$$

The proof performs case analysis on the universally quantified x. In the case $x = 0$, the goal becomes $0 + 0 \equiv 0$. The system reasons automatically that $0 + 0 \overset{\iota}{=} 0$, and so the proof term `Refl` is able to discharge the goal, which is equivalent to $0 \equiv 0$ modulo computation. In the case $x = 1 + x'$ for some x', the goal becomes $(1 + x') + 0 \equiv 1 + x'$, which the system automatically converts to $1 + (x' + 0) \equiv 1 + x'$ via computation. The `rewrite` command explicitly rewrites the goal using the inductive hypothesis, that is $x' + 0 \equiv x$, resulting in the goal $1 + x' \equiv 1 + x'$, which is directly provable by the reflexivity judgment `Refl`.

2.3.2 Embedding Classical Powersets

A common occurrence in mathematics is to represent the set of predicates, or classifications, over some set A as the powerset $\wp(A)$. In ITT, these powersets are represented directly as their characteristic functions $\phi \;:\; A \to \mathit{prop}$. Because these characteristic functions can be undecidable in general, there is little that can be done with powersets other than construct other powersets. This leads to the powerset $\wp(A) \;\coloneqq\; A \to \mathit{prop}$ behaving as a modality in ITT which can't be escaped from.

EXAMPLE Consider the set of of integers which are odd. Classically this is represented:

$$\textit{odd-integers} \in \wp(\mathbb{Z}) \;\coloneqq\; \{z \mid \textit{odd}(z)\}$$

However, In ITT this set is represented directly as its characteristic function $\phi \coloneqq \textit{odd} : \mathbb{Z} \to \textit{prop}$.

Next consider the following classical function which classifies an arbitrary set of integers using an abstract description:

$$\textit{classify} \;\in\; \wp(\mathbb{Z}) \to \{\texttt{all-even}, \texttt{all-odd}, \texttt{empty}, \texttt{even-and-odd}\}$$

$$\textit{classify}(I) \;\coloneqq\; \begin{cases} \texttt{all-even} & \textit{if} \quad \exists i \in I \wedge \forall i \in I.\, \textit{even}(i) \\ \texttt{all-odd} & \textit{if} \quad \exists i \in I \wedge \forall i \in I.\, \textit{odd}(i) \\ \texttt{empty} & \textit{if} \quad I = \varnothing \\ \texttt{even-and-odd} & \textit{if} \quad \exists i_1, i_2 \in I.\, \textit{even}(i_1) \wedge \textit{odd}(i_2) \end{cases}$$

This function is not representable in ITT because it requires *computing* the abstract description given an arbitrary set of integers. However, this function is definable by mapping to a singleton powerset:

$$\textit{classify} \;:\; \wp(\mathbb{Z}) \to \overset{1}{\wp}(\{\texttt{all-even}, \texttt{all-odd}, \texttt{empty}, \texttt{even-and-odd}\})$$

$$\textit{classify}(I) \;\coloneqq\; \bigcup \begin{cases} \{\texttt{all-even} & \mid \quad \exists i \in I \wedge \forall i \in I.\, \textit{even}(i)\} \\ \{\texttt{all-odd} & \mid \quad \exists i \in I \wedge \forall i \in I.\, \textit{odd}(i)\} \\ \{\texttt{empty} & \mid \quad I = \varnothing\} \\ \{\texttt{even-and-odd} & \mid \quad \exists i_1, i_2 \in I.\, \textit{even}(i_1) \wedge \textit{odd}(i_2)\} \end{cases}$$

This is a mapping between specifications, and is perfectly definable in ITT. The escape-hatch used here is the fact that singleton powersets $\overset{1}{\wp}(A)$ are not isomorphic to the underlying carrier A in a constructive setting (no mapping exists for $\overset{1}{\wp}(A) \to A$), whereas in a classical setting $\overset{1}{\wp}(A)$ and A are isomorphic and interchangeable.

2.3.3 Embedding General Classical Reasoning

Although variants of ITT do *not* allow direct definitions of the Law of Excluded Middle (LEM), they *do* support defining LEM and all other classical logical formulas via an embedding called *double negation*. This embedding serves as a modality in the logic which explicitly separates proofs which carry computational content (*i.e.* constructive) from those that don't (*i.e.* classical), while still supporting fully general mathematical reasoning.

Because of the existence of the double negation embedding, it would be incorrect to say ITT supports fewer theorems than a non-constructive higher-order logic. Terms embedded in the double negation type are still manifestations of "truth."

Example Consider the law of excluded middle, as stated classically:

$$\forall p \in \mathit{prop}.\ p \vee \neg p \tag{LEM}$$

The direct embedding of this proposition in ITT is a dependent product:

$$\prod_{p\ :\ \mathit{prop}} p \vee \neg p \tag{LEM-ITT}$$

the proof of which in ITT would be a dependent function:

$$\lambda p\ :\ \mathit{prop}.\ (\ldots\ :\ p \vee \neg p) \tag{LEM-ITT-TERM}$$

However, the constructive interpretation of λ-terms prohibit such a definition for arbitrary propositions, which are not always decidable. For example, consider instantiating (LEM-ITT) with the proposition "the Nth turing machine halts." The

hypothetical (LEM-ITT-TERM) would then have to *compute* in finite time whether or not the proposition is true, however it is well known that this particular proposition is not decidable in general, hence there is no term which can inhabit (LEM-ITT).

LEM *can*, however, be embedded in ITT in a way which explicitly discards the constraint that it carries computational content. The embedding is that of double negation, so:

$$\prod_{p\,:\,prop} \neg\neg(p \vee \neg p)) \qquad \text{(LEM-ITT-DN)}$$

where the definition of negation is:

$$\neg p \;:\equiv\; p \to \mathit{false}$$

The proposition (LEM-ITT-DN) is then a refutation of terms which claim to refute LEM. This proposition is inhabited in ITT:

$$\lambda p\;:\;\mathit{prop}.\,\lambda X\;:\;\neg(p \vee \neg p).\,X(\mathtt{Inr}(\lambda x\;:\;p.\,X(\mathtt{Inl}(x))))$$

2.3.4 Conclusion

In this section we reviewed mechanized verification as achieved through Intuitionistic Type Theory (ITT). This was done through a discussion of how ITT is formulated as a logic where constructions carry computational content, which are then extracted and executed as certified programs. We then followed with discussions of equality and embedding classical notions like powersets and the Law of Excluded Middle (LEM), as well as supporting examples.

Portions of this thesis rely on mechanized verification using the Agda proof assistant, based on CIC, a descendent in the ITT family of logics. The computational nature of ITT is crucial in our use of these tools, as we extract certified programs not only from algorithms directly embedded in Agda, but we extract programs directly from *proofs* as well. The ability to extract programs directly from proofs is a well known feature of ITT, but has yet to be realized in the setting of calculational abstract interpretation until the results presented in this thesis.

Chapter 3: Technical Overview

3.1 Constructive Galois Connections

Galois connections are a foundational tool for structuring abstraction in semantics and their use lies at the heart of the theory of abstract interpretation. Yet, mechanization of Galois connections remains limited to restricted modes of use, preventing their general application in mechanized metatheory and certified programming.

We present Constructive Galois Connections [Darais and Van Horn, 2016], a variant of Galois connections that is effective both on paper and in proof assistants; is complete w.r.t a large subset of classical Galois connections; and enables more general reasoning principles, including the "calculational" style advocated by Cousot.

To design constructive Galois connection we identify a restricted mode of use of classical ones which is both general and amenable to mechanization in dependently-

typed functional programming languages. Crucial to the metatheory is the addition of monadic structure to Galois connections to control a "specification effect." Effectful calculations may reason classically, while pure calculations have extractable computational content. Explicitly moving between the worlds of specification and implementation is enabled by the metatheory.

To validate the approach we provide two case studies in mechanizing existing proofs from the literature: one uses calculational abstract interpretation to design a static analyzer [Cousot, 1999], the other forms a semantic basis for gradual typing [Garcia et al., 2016]. Both mechanized proofs closely follow their original paper-and-pencil counterparts, employ reasoning principles not captured by previous mechanization approaches [Monniaux, 1998, Pichardie, 2005], support the extraction of verified algorithms, and are novel.

3.1.1 The Problem

The issue with the classical Galois connection framework is that some functions cannot be defined constructively, and the consequence of this is that definitions which use Galois connections cannot always be extracted as verified algorithms.

To illustrate the problems with mechanizing classical Galois connections, consider a simple parity program analyzer designed using the following Galois connection

between natural numbers $\mathbb{N}$ and parities $\mathbb{P}$ (plus Galois Connection laws not shown):

$$\begin{array}{rcl} \mathbb{P} & \coloneqq & \{\text{EVEN}, \text{ODD}\} \\ \alpha & : & \wp(\mathbb{N}) \to \wp(\mathbb{P}) \\ \gamma & : & \wp(\mathbb{P}) \to \wp(\mathbb{N}) \end{array} \qquad \begin{array}{rcl} \alpha(N) & \coloneqq & \bigcup\limits_{n \in N} \begin{cases} \{\text{EVEN}\} & \textit{if} \quad \textit{even}(n) \\ \{\text{ODD}\} & \textit{if} \quad \textit{odd}(n) \end{cases} \\ \gamma(P) & \coloneqq & \bigcup\limits_{p \in P} \begin{cases} \{n \mid \textit{even}(n)\} & \textit{if} \quad p = \text{EVEN} \\ \{n \mid \textit{odd}(n)\} & \textit{if} \quad p = \text{ODD} \end{cases} \end{array}$$

A program analyzer $\mathcal{A} : \wp(\mathbb{P}) \to \wp(\mathbb{P})$ for a concrete semantics $\mathcal{C} : \wp(\mathbb{N}) \to \wp(\mathbb{N})$ is then justified by relating to the composition of $\mathcal{C}$ with α and γ:

$$\alpha \circ \mathcal{C} \circ \gamma \sqsubseteq \mathcal{A} \tag{Soundness}$$

The trouble in mechanizing (Soundness) is that $\mathcal{A}$ is expected to be computable, meaning its type $\wp(\mathbb{P}) \to \wp(\mathbb{P})$ represents an *algorithm* mapping between finite sets of parities. However, the specification $\alpha \circ \mathcal{C} \circ \gamma$, also at type $\wp(\mathbb{P}) \to \wp(\mathbb{P})$, represents an induced *specification* which cannot in general be computed.

In a constructive setting, these two powerset types have different representations. Constructed powersets $\wp(\mathbb{P})$ are modeled with a datatype like a list or binary tree, or in the case of $\wp(\mathbb{P})$ as an enumeration of its inhabitants:

$$\wp(\mathbb{P}) \approx \mathbb{P}^+ \coloneqq \{\bot, \text{EVEN}, \text{ODD}, \top\}$$

However, specification powersets $\wp(\mathbb{P})$ are modeled as predicates on $\mathbb{P}$:

$$\wp(\mathbb{P}) \approx \mathbb{P} \to \mathit{prop}$$

On paper the encoding of $\wp(\mathbb{P})$ doesn't matter, but to perform verified program extraction on $\mathcal{A}$, a solution must be found for encoding proofs like *sound* which transition between specification and algorithm.

The current state-of-the art in mechanized abstract interpretation is to only embed γ in the proof assistant, because α is the problematic mapping w.r.t. constructivity. This results in so-called γ-*only* definitions and proofs, for example (Soundness) has an equivalent formulation using only γ:

$$\mathcal{C} \circ \gamma \sqsubseteq \gamma \circ \mathcal{A} \qquad \text{(Soundness [}\gamma\text{-only])}$$

However, this approach doesn't allow for the calculational approach to abstract interpretation, where the very definition of $\mathcal{A}$ is derived directly from its induced specification $\alpha \circ \mathcal{C} \circ \gamma$.

3.1.2 The Main Ideas

We develop constructive Galois connections from the insight that many classical Galois connections used in practice are of a particular restricted form, which is reminiscent of a direct-style verification. Constructive Galois connections are the general abstraction theory for this setting and can be mechanized effectively.

We observe that constructive Galois connections contain monadic structure which isolates classical specifications from constructive algorithms. Within the effectful fragment, all of classical Galois connection reasoning can be employed, while within the pure fragment, functions must carry computational content. Remarkably, calculations can move between these modalities and verified programs may be extracted from the end result of calculation.

Constructive Galois Connections Our constructive theory of Galois connections can be seen as a restricted mode of use of classical Galois connections. The essence of the theory is a different adjunction η/μ instead of α/γ:

$$\begin{array}{ll} \eta \ : \ \mathbb{N} \to \mathbb{P} & \eta(n) \coloneqq \begin{cases} \textsc{even} & \mathit{if} \quad \mathit{even}(n) \\ \textsc{odd} & \mathit{if} \quad \mathit{odd}(n) \end{cases} \\ \mu \ : \ \mathbb{P} \to \wp(\mathbb{N}) & \mu(p) \coloneqq \begin{cases} \{n \mid \mathit{even}(n)\} & \mathit{if} \quad p = \textsc{even} \\ \{n \mid \mathit{odd}(n)\} & \mathit{if} \quad p = \textsc{odd} \end{cases} \end{array}$$

along with the adjunction correspondence:

$$n \in \mu(p) \iff \eta(n) \sqsubseteq p \qquad \text{(CGC-Corr)}$$

In this restricted theory, executable algorithms can be extracted directly from the results of proofs in the abstract interpretation paradigm. This setting supports all the benefits of a general abstraction framework like classical Galois connections: synthesized specifications, soundness and completeness properties, and even calculational derivations of program analyzers.

Classical Galois connections can be recovered from constructive Galois through a lifting:

$$\begin{array}{ll} \alpha \ : \ \wp(\mathbb{N}) \to \wp(\mathbb{P}) & \alpha(N) \coloneqq \{\eta(n) \mid n \in N\} \\ \gamma \ : \ \wp(\mathbb{P}) \to \wp(\mathbb{P}) & \gamma(P) \coloneqq \bigcup_{p \in P} \mu(p) \end{array}$$

as well as the classical Galois connection correspondence:

$$N \subseteq \gamma(P) \iff \alpha(N) \subseteq P$$

We also demonstrate that when a constructive Galois connection exists underneath a classical Galois connection, all properties which could be proved in the classical

setting can likewise be proved in the constructive setting, which results in much simpler proofs.

THE SPECIFICATION EFFECT We call the powerset type $\wp(A)$ a specification effect because it has monadic structure, supports encoding arbitrary properties over values in A, and cannot be "escaped from" in constructive logic, similar to the IO monad in Haskell. In classical mathematics, there is an isomorphism between singleton powersets $\wp^1(A)$ and the set A. However, no such constructive mapping exists for $\wp^1(A) \to A$. Such a function would decide arbitrary predicates in $A \to \mathit{prop}$ to *compute* the A inside the singleton set. This observation, that you can program inside $\wp(\_)$ monadically in constructive logic, but you can't escape the monad, is why we call it a specification effect.

The soundness and completeness conditions generated by constructive Galois connections come from a monadic adjunction, and are therefore recast in a monadic setting. For example, the constructive equivalent to (Soundness) is:

$$\mathit{pure}(\eta) \circledast \mathcal{C} \circledast \mu \sqsubseteq \mathcal{A} \qquad \text{(Soundness } \eta/\mu\text{)}$$

Both sides of the equation have type $\mathbb{P} \to \wp(\mathbb{P})$, the monadic interpretation of which is "a function from $\mathbb{P}$ to $\mathbb{P}$ which performs specification effects." This is empowering because it allows one to be explicit about the induced $\mathit{pure}(\eta) \circledast \mathcal{C} \circledast \mu$ being a specification, meaning it has effects, and the analysis $\mathcal{A}$ being an algorithm, meaning it has no effects. One can even derive the definition of $\mathcal{A}$ from this specification, and in the process eliminate the "specification effect" through program calculation, the

results of which can immediately be extracted and executed.

3.1.3 Evaluation

To support the utility of our theory we build a library for constructive Galois connections in Agda [Norell, 2007] and mechanize two existing abstract interpretation proofs from the literature. The first is drawn from Cousot's monograph [1999], which derives a correct-by-construction analyzer from a specification induced by a concrete interpreter and Galois connection. The second is drawn from Garcia et al.'s *Abstracting Gradual Typing* [2016], which uses abstract interpretation to derive static and dynamic semantics for gradually typed languages from traditional static types. Both proofs use the calculational style of abstract interpretation which is not handled by prior mechanization approaches. The mechanized proofs closely follow the original pencil-and-paper proofs, which use both abstraction and concretization, while still enabling the extraction of certified algorithms. Neither of these papers have been previously mechanized. Moreover, we know of no existing mechanized proof involving calculational abstract interpretation.

Finally, we develop the metatheory of constructive Galois connections, prove them sound, and make precise their relationship to classical Galois connections. The metatheory is also itself mechanized in Agda.

3.2 Galois Transformers

The design and implementation of static analyzers has become increasingly systematic. Yet although the design is systematic, implementing an analyzer and proving it sound remains a tedious and error prone effort. The issue is that static analysis features and their proofs of soundness do not compose well, preventing reuse in both implementation and metatheory. Due to the lack of compositional components, small changes to an analyzer's design often require large changes to its implementation and proof.

We solve the problem of constructing static analyzers and their proofs from reusable components by introducing Galois Transformers [Darais et al., 2015]: monad transformers that transport Galois connection properties. In concert with a monadic interpreter, we define analysis parameters that implement building blocks for classic analysis features like context, object, heap, path and flow (in)sensitivity. Each component comes with modular proofs and is defined independently of a particular programming language semantics.

Significantly, Galois transformers are proven sound once and for all, making them truly reusable analysis components. As new analysis features and abstractions are developed and mixed in, soundness proofs need not be reconstructed, as the composition of a monad transformer stack is sound by virtue of its constituents. Galois transformers provide a viable foundation for reusable and composable metatheory for program analysis, and are amenable to mechanized verification with proof assistants.

Finally, Galois transformers shift the level of abstraction in analysis design and

implementation. Using Galois transformers, non-specialists are able to synthesize sound analyzers over a number of parameters, which can then be then be tuned in plug-and-play fashion to recovering a wide range of analyses. Tuning parameters in our framework requires no change to the implementation or proof of correctness, which enables rapids prototyping of the analyzer design space.

3.2.1 The Problem

The problem with current approaches to program analysis design is they are unable to account for path and flow sensitivity as a parameter to the analysis, both in implementation and proof.

To illustrate path and flow sensitivity, consider verifying the absence of division by zero errors in the following program:

```
(1)  function example(i : int) → int
(2)    var x, y : int
(3)    if i ≠ 0
(4)      then x := 0
(5)      else x := 1
(6)    if i ≠ 0
(7)      then y := 100/i
(8)      else y := 100/x
(9)    return y
```

This program branches on the function argument i to define x such that $i \neq 0 \Leftrightarrow x = 0$ (and therefore $i = 0 \Leftrightarrow x \neq 0$) in lines 3–5. The goal of the analysis is to discover division by zero errors, and the two divisions at lines 7 and 8 are always safe because

of the above correlation between x and i.

To verify the above program as free of division by zero errors, a *Path Sensitive* (PS) analysis is required, which is computationally expensive. A less precise but more performant design is a *Flow Sensitive* (FS) analysis, which is only able to verify the first division at line 7. Finally, a *Flow Insensitive* (FI) analysis is the least precise design choice, is unable to verify either of the divisions, but is far more performant that a PS or FS design.

These three variations of analysis—path sensitive (PS), flow sensitive (FS) and flow insensitive (FI)—are strictly ordered in terms of precision:

$$\mathit{prec}(\text{PS}) > \mathit{prec}(\text{FS}) > \mathit{prec}(\text{FI})$$

and inversely ordered in terms of both average and worst-case performance:

$$\mathit{perf}(\text{FI}) > \mathit{perf}(\text{FS}) > \mathit{perf}(\text{PS})$$

In security-critical settings, path sensitivity is often the right choice despite the added cost. However, in performance-critical settings, path sensitivity is infeasible because of its cost, which suggests using a flow sensitive or flow-insensitive analysis. In order to rapidly prototype this design space to find the best fit for a particular application, the path and flow sensitivity of the analyzer must be compartmentalized and supplied as a parameter.

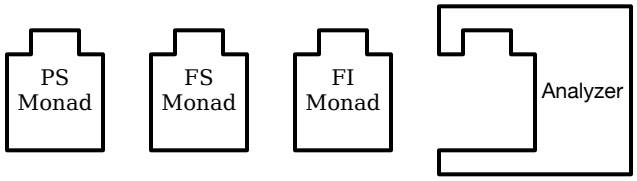
Previous approaches to program analysis require rewriting large parts of the design to support each variant of path and flow sensitivity. The issue is magnified in the setting of mechanized verification, where rewriting an implementation means

rewriting a proof, and where the proof effort of a development is much more costly than that of a pencil-and-paper formalization.

3.2.2 The Main Ideas

Galois transformers are reusable building blocks for building analysers that supply each choice in the path sensitivity spectrum: flow insensitive, flow sensitive and path sensitive. A flow insensitive Galois transformers can simply be replaced by a path sensitive Galois transformer, requiring no further change to the analyzer or its proof. In this way, one can rapidly prototype the path and flow sensitivity design space for a particular program analysis. Proofs of correctness for the analyzer also carry over between different instantiations of Galois transformers.

Galois transformers support rapidly prototyping choices in path and flow sensitivity by introducing a novel parameter to the program analyzer: the *monad* used for executing the analysis. Monads are introduced in the analyzer design to capture the interaction between analysis results (like $x \neq 0$) and nondeterministic branching in the analyzer (like analyzing `if` $x = 5$ `then` y `else` z when $x \neq 0$). By changing the monad, and therefore how analysis results and nondeterminism interact, we recover each of PS, FS and FI implementations for the analyzer.

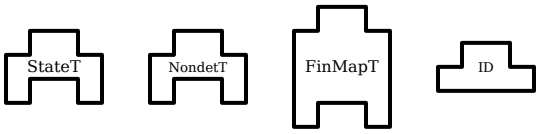


Designing Parameterized Analyzers To design an analyzer parameterized by a monad, one first identifies the parts of the analysis which communicate analysis results and the parts which branch due to nondeterminism. Rather than implement these parts of the analysis directly, the analyzer is instead written using a *monadic effect* interface consisting of state and nondeterminism effects. The state effect is used for manipulating analysis results, and the nondeterminism effect is used for branching. The analyzer can be executed only after instantiating its monad parameter with some monad that implements state and nondeterminism effects. Most importantly, a monadic analyzer can be proven correct using monad and monad effect laws, independent from a particular monad instantiation.

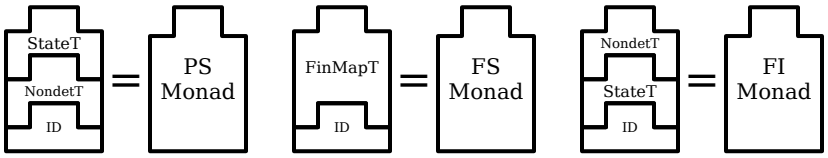
Constructing Monad Parameters To help construct monads for a parameterized analyzer we design a library of *monad transformers*. Monad transformers are compositional building blocks for constructing monads. The monad transformers used to construct a monad, as well as their order of assembly, determine the path and flow sensitivity properties of the analysis.

We identify three monads for use in our setting: the state monad transformer (StateT) which implements state effects, the nondeterminism monad transformer (NondetT) which implements nondeterminism effects, and the finite map monad transformer (FinMapT) which implements both nondeterminism and state effects. [Each of StateT, NodnetT and FinMapT are generally useful, even outside our application to program analyzers. StateT is standard from the literature [Liang et al., 1995, Moggi, 1989], and NondetT and FinMapT are novel in this work.] These

monad transformers can be assembled in any order, and use the identity monad (ID) as the starting point of composition.



Enumerating the combinations of monad transformers which implement both state and nondeterminism results in PS, FS and FI monads. When plugged into a parameterized interpreter the result is a path sensitive, flow sensitive and flow insensitive program analyzer respectively.



Furthermore we show that these monad transformers also propagate Galois connections, which is essential for achieving modularity in the soundness proofs for a parameterized analyzer. We call these monad transformers *Galois transformers* because of their Galois connection properties.

3.2.3 Evaluation

We evaluate Galois transformers by proving key metatheory properties of end-to-end static analysis verification, and by implementing a Galois transformers library and prototype client analysis in Haskell.

End-to-end Correctness The end-to-end correctness of a static analyzer in our setting is justified using compositional components:

1. a proof that the monadic interpreter recovers the concrete semantics,
2. a proof that the monadic interpreter is monotonic, and
3. a proof of abstraction between concrete and abstract monad parameters.

The user of our framework is responsible for (1) and (2). We prove that (3) is synthesized by the properties of Galois transformers, in addition to the implication that (1–3) yields a sound program analysis.

3.3 Abstracting Definitional Interpreters

Two dominant schools of thought for designing program analyzers are the constraint-based approach and small-step state-machines-based approach. In each paradigm (respectively), the analysis is computed by the least-fixed-point of a set of constraint equations, or through reachability of a relational small-step collecting semantics.

On the other hand, there is a large body of work on denotational semantics and definitional interpreters, or so-called "big-step" interpreters. Definitional interpreters are popular for describing concrete semantics because they are compositional by nature. However, definitional interpreters have not seen adoption for describing abstract semantics, or as the basis for defining program analyzers, particularly in the higher order setting.

To bridge this gap we develop Abstract Definitional Interpreters [Darais et al.,

2017] and show that definitional interpreters written in monadic style can express not only the usual notion of (concrete) interpretation, but also a wide variety of collecting semantics, abstract interpretations, symbolic execution, and their intermixings.

In this work we reconstruct a definitional abstract interpreter for a higher-order language to use monadic operations and a novel fixpoint iteration strategy. Through a monadic definitional design, we achieve a computable abstract interpreter that arises from the composition of simple, independent components.

Remarkably, the resulting program analyzer implements a form of pushdown control flow analysis (PDCFA) in which calls and returns are always properly matched in the abstract semantics. True to the definitional style of Reynolds, the evaluator involves no explicit mechanics to achieve this property; it is simply inherited from the defining language.

3.3.1 The Problem

The challenge when using definitional interpreters as a foundation for program analysis is the treatment of fixpoints. For a definitional interpreter, the meaning of fixpoints in the object language is inherited from the metalanguage. This is problematic when metalanguage fixpoints involve nontermination, which prevents obtaining a computable program analysis.

Another challenge with definitional interpreters is they omit description of

intermediate execution configurations. For example, consider this program:

```
function loop() → void
  var x := 42
  while(true)
    skip
```

The concrete denotation of calling *loop* is $\bot$, which does not mention intermediate facts about the program, like $x = 42$. Small-step and constraint-based approaches support analysis of intermediate results because they are by-nature explicit about reachable program configurations.

When tuning the precision of a program analysis, a challenging point of the design is approximating the call-and-return structure of program execution. To illustrate this, consider analyzing the following program:

```
(1)  function id(x : any) → any
(2)    return x
(3)  function main() → void
(4)    var y := id(1)
(5)    print("Y")
(6)    var z := id(2)
(7)    print("Z")
```

The printed output of this program is `"YZ"`. However, most control-flow analyzers will report that the output could be any string that matches the regular expression `"Y*Z"`. The problem is that control flow analyzers construct a global graph of call edges, in this case from lines 5 and 7 to the body of *id*, and return edges, in this case from *id* back to lines 5 and 7. Without precise call-return matching, control-flow analyzers get confused and think the program could call *id* at line 5 and

then return to line 7, or call *id* at line 7 and return to line 5. A "k-call-site-sensitive" analysis can distinguish these cases, but only up to a finite call-depth.

A pushdown analysis solves the call/return matching problem up to infinite depth. Prior descriptions of pushdown analysis are set in the context of actual pushdown automata [Reps et al., 1995], Dyck state graphs [Earl et al., 2012] or small-step state machines [Gilray et al., 2016b, Johnson and Van Horn, 2014, Vardoulakis and Shivers, 2010], and each approach requires ad-hoc extensions and instrumentation to the design of the program analyzer.

3.3.2 The Main Ideas

Our key insights are to design definitional interpreters in monadic, open-recursive style, and to design a novel fixpoint algorithm tailored specifically to the setting of higher-order definitional interpreters. The extensible nature of the interpreter allows us to recover a wide-range of analyses through its instantiation, including widening techniques, precision preserving abstractions, and symbolic execution for program verification.

We also realize a new technique for defining abstract interpreters with pushdown precision, meaning the analysis precisely matches function calls to returns. In the setting of definitional interpreters, this property is inherited from the defining metalanguage and requires no instrumentation to the analysis at all.

Unfixing Interpreters To support finding fixpoints for definitional interpreters, we first design definitional interpreters in open-recursive style. For example, the

denotation of an `if` expression is written:

$$
\begin{aligned}
&E \;:\; (exp \to val) \to exp \to val \\
&\ldots \\
&E(E')(\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3) \coloneqq \\
&\quad \texttt{if}\ E'(e_1) \stackrel{?}{=} \texttt{True then}\ E'(e_2)\ \texttt{else}\ E'(e_3) \\
&\ldots
\end{aligned}
$$

The standard evaluator is then recovered by $Y(E)$, and we allow abstract evaluators to be defined through (total) approximating fixpoint finding functions.

To find fixpoints for abstract definitional interpreters, we design a novel caching algorithm ($Y^\sharp$), which when applied to an unfixed interpreter yields a sound and computable analysis. To support abstract fixpoints, we redesign the unfixed evaluator to consume and output a cache so it can communicate with the fixpoint algorithm.

(1) $Y^\sharp \;:\; ((exp \times cache \to val \times cache) \to exp \times cache \to val \times cache)$
(2) $\quad \to exp \to cache$
(2) $Y^\sharp(F)(e) \coloneqq lfp(\lambda\$^o.$
(3) $\quad \texttt{let rec}\ E \coloneqq F(\lambda\langle e, \$^I\rangle.$
(4) $\quad\quad \texttt{if}\ e \in \$^I\ \texttt{then}\ \langle \$^I(e), \$^I\rangle\ \texttt{else}$
(5) $\quad\quad\quad \texttt{let}\ \langle v, \$^{I\prime}\rangle \coloneqq E(e, \$^I[e \mapsto \$^O(e)])$
(6) $\quad\quad\quad \texttt{in}\ \langle v, \$^{I\prime}[e \mapsto v]\rangle)$
(7) $\quad \texttt{in}\ \pi_2(E(e)))$

The algorithm computes the least-fixed-point of a cache ($\$^o$) which is computed by calling the unfixed evaluator (F) and intercepting recursive calls (at $(e, \$^I)$) to either not repeat work if it has already been done (line 4), record the current configuration so as to not loop in the future (line 5), and record the results of evaluation in the cache (line 6).

Monadic Definitional Interpreters To support a multitude of different analyzers from a single definitional interpreter, we write the open-recursive evaluator in monadic style, so the above example becomes:

$$
\begin{aligned}
&E \;:\; (exp \to M(val)) \to exp \to M(val) \\
&\ldots \\
&E(E')(\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3) \coloneqq \texttt{do} \\
&\quad v_1 \leftarrow E'(e_1) \\
&\quad \texttt{if}\ v_1 \stackrel{?}{=} True\ \texttt{then}\ E'(e_2)\ \texttt{else}\ E'(e_3) \\
&\ldots
\end{aligned}
$$

Different monads can then be plugged into the evaluator to recover different analyzers. The monadic abstraction is also essential to treat the cache state-passing version of the evaluator in a systematic way, as just another cell of monadic state.

Inheriting Pushdown Precision Small-step methods to programming language semantics must model the context of evaluation, either through syntactic evaluation contexts or stack frames. Perfect stack precision is lost in this approach because stack frames are modeled explicitly, and the process of approximation is applied naively to the model of the stack.

We observe that perfect stack precision is already present in the definitional interpreter, and therefore yields a pushdown analysis, even when executed as an approximating abstract interpreter. In the case of definitional interpreters, the evaluation context is implicit in the call-and-return semantics of the defining programming language, which is already perfectly precise. Because no approximation is made in the model for evaluation contexts, the resulting abstraction for evaluation contexts

is perfectly precise.

3.3.3 Evaluation

We implement a general framework of abstract definitional interpreters in Racket and recover various abstract interpreters, including various widening techniques, a mixed concrete/abstract abstraction for arithmetic expressions, and a symbolic executor which can perform program verification.

We prove the approach sound w.r.t. a derived big-step semantics, where the key insight in the formalism is to model not only standard big-step *evaluation* relations, but also big-step *reachability* relations, which we carry out through each of concrete, collecting, and abstract semantics.

The formalism begins with a big-step semantics ($\rho \vdash e, \sigma \Downarrow \sigma'$) augmented with big-step reachability ($\rho \vdash e, \sigma \Uparrow \varsigma$) which describes reachable configurations ς. We show a combination of these relations ($[\![e]\!]^{bs}$) forms a "complete" big-step semantics in that it contains the same information as the small-step setting ($[\![e]\!]^{ss}$). We then perform systematic abstraction of the complete big-step semantics ($[\![e]\!]^{bs}$) and justify computing analysis solutions as the least-fixpoint of a cache which simulates both big-step evaluation and reachability.

Chapter 4: Constructive Galois Connections

4.1 Introduction

Abstract interpretation is a general theory of sound approximation widely applied in programming language semantics, formal verification, and static analysis [Cousot and Cousot, 1976, 1977, 1979, 1992, 2014]. In abstract interpretation, properties of programs are related between a pair of partially ordered sets: a concrete domain, $\langle \mathcal{C}, \sqsubseteq \rangle$, and an abstract domain, $\langle \mathcal{A}, \preceq \rangle$. When concrete properties have a $\preceq$-most precise abstraction, the correspondence is a *Galois connection*, formed by a pair of mappings between the domains known as *abstraction* $\alpha \in \mathcal{C} \mapsto \mathcal{A}$ and *concretization* $\gamma \in \mathcal{A} \mapsto \mathcal{C}$ such that $c \sqsubseteq \gamma(a) \iff \alpha(c) \preceq a$. Since its introduction by Cousot and Cousot in the late 1970s, this theory has formed the basis of static analyzers, type systems, model-checkers, obfuscators, program transformations, and many more applications [Cousot, 2008].

Given the remarkable set of tools contributed by this theory, an obvious desire is to incorporate its use into proof assistants to mechanically verify proofs by abstract interpretation. When embedded in a proof assistant, verified algorithms such as static analyzers can then be extracted from these proofs.

Monniaux first achieved mechanization for the theory of abstract interpretation

with Galois connections in Coq [1998]. However, he notes that the abstraction (α) side of Galois connections is problematic since it requires the admission of non-constructive axioms. Use of these axioms prevents the extraction of certified programs. So while Monniaux was able to mechanically verify proofs by abstract interpretation in its full generality, certified artifacts could not extracted in general.

Pichardie subsequently tackled the extraction problem by using a restricted formulation of abstract interpretation that relied only on the concretization (γ) side of Galois connections [2005]. Doing so avoids the use of axioms and enables extraction of certified artifacts. This technique is effective and has been used to construct several certified static analyzers [Barthe et al., 2007, Blazy et al., 2013, Cachera and Pichardie, 2010, Pichardie, 2005], most notably the Verasco static analyzer, part of the CompCert C compiler [Jourdan et al., 2015, Leroy, 2009]. Unfortunately, this approach sacrifices the full generality of the theory. While in principle the technique could achieve mechanization of existing soundness *theorems*, it cannot do so faithful to existing *proofs*. In particular, Pichardie writes [2005, p. 55]:[1]

> The framework we have retained nevertheless loses an important property of the standard framework: being able to derive a correct approximation $f^\sharp$ from the specification $\alpha \circ f \circ \gamma$. Several examples of such derivations are given by Cousot [1999]. It seems interesting to find a framework for this kind of symbolic manipulation, while remaining easily formalizable in Coq.

[1] Translated from French by the present author.

This important property is the so-called "calculational" style, where an abstract interpreter ($f^\sharp$) is derived in a correct-by-construction manner from a concrete interpreter (f) composed with abstraction and concretization ($\alpha \circ f \circ \gamma$). This calculational method is detailed in Cousot's monograph [1999], which concludes:

> The emphasis in these notes has been on the correctness of the design by calculus. The mechanized verification of this formal development using a proof assistant can be foreseen with automatic extraction of a correct program from its correctness proof.

In the subsequent 17 years, this vision has remained unrealized, and clearly the paramount technical challenge in achieving it is obtaining both *generality* and *constructivity* in a single framework.

In this chapter we contribute *constructive Galois connections*, a framework for abstract interpretation with Galois connections that achieves both generality and constructivity, thereby enabling calculational style proofs which make use of both abstraction (α) and concretization (γ), while also maintaining the ability to extract certified static analyzers. We develop constructive Galois connections from the insight that many classical Galois connections used in practice are of a particular restricted form—which is reminiscent of a direct-style verification—and that this restricted form both supports calculation and is amenable to mechanized verification. Constructive Galois connections are the general abstraction theory for this restricted setting of classical Galois connections.

We observe that constructive Galois connections contain monadic structure

which isolates classical specifications from constructive algorithms. Within the effectful fragment, all of classical Galois connection reasoning can be employed, while within the pure fragment, functions must carry computational content. Remarkably, calculations can move between these modalities and verified programs may be extracted from the end result of calculation, which must be "effect-free."

To support the utility of our theory we build a library for constructive Galois connections in Agda [Norell, 2007] and mechanize two existing abstract interpretation proofs from the literature. The first is drawn from Cousot's monograph [1999], which derives a correct-by-construction analyzer from a specification induced by a concrete interpreter and Galois connection. The second is drawn from Garcia et al.'s *Abstracting Gradual Typing* [2016], which uses abstract interpretation to derive static and dynamic semantics for gradually typed languages from traditional static types. Both proofs use the "important property of the standard framework" identified by Pichardie, which is not handled by prior mechanization approaches. The mechanized proofs closely follow the original pencil-and-paper proofs, which use both abstraction and concretization mappings, while still enabling the extraction of certified algorithms. Neither of these papers have been previously mechanized. Moreover, we know of no existing mechanized proof involving calculational abstract interpretation.

Next, we develop the metatheory of constructive Galois connections, prove they are sound and complete, and make precise their relationship to classical Galois connections. The metatheory is itself mechanized; claims are marked with "AGDA✓" whenever they are proved in Agda. (All claims are marked.)

Finally, we explore the relationship between classical and constructive Galois connections in much more detail. We do this through defining constructive analogs to classical Galois connection primitive and connectives, and through two examples drawn from our first case study which we work out in full detail. Through these extended examples, we compare and contrast the differences between abstraction-directed and concretization-directed calculations, and between sound and complete calculations, for both classical and constructive styles. The outcome of this study is a better understanding of how constructive calculations interact with classical Galois connections, how the mechanics of optimality changes between frameworks, and how to calculate multivalued algorithms in the constructive setting.

Contributions This chapter contributes the following:

- A foundational theory of constructive Galois connections which is both general and amenable to mechanization using a dependently typed functional programming language;
- A proof library and two case studies from the literature for mechanized abstract interpretation; and
- The first mechanization of calculational abstract interpretation; and
- A detailed discussion of the relationship between constructive and classical Galois connections, and their interaction.

The remainder of the chapter is organized as follows. First we give a tutorial on verifying a simple analyzer from two different perspectives: direct verification

(§ 4.2.1) and abstract interpretation with Galois connections (§ 4.2.2), highlighting mechanization issues along the way. We then present constructive Galois connections as a marriage of the two approaches (§ 4.3). We provide two case studies: the mechanization of an abstract interpreter from Cousot's calculational monograph (§ 4.4), and the mechanization of Garcia, Clark and Tanter's work on gradual typing *via* abstract interpretation (§ 4.5). Next, we formalize the metatheory of constructive Galois connections (§ 4.6), define constructive analogs of common classical Galois connection primitives and connectives (§ 4.7), and work through two extended examples in detail: the first to compare and contrast calculation styles (§ 4.8) and discuss deriving optimal interpreters (§ 4.9), and the second to explore multivalued constructive calculations (§ 4.10). Finally, we relate our work to the literature (§ 4.11), and conclude (§ 4.12).

4.2 Verifying a Simple Static Analyzer

In this section we contrast two perspectives on verifying a static analyzer: using a direct approach, and using the theory of abstract interpretation with Galois connections. The direct approach is simple but lacks the benefits of a general abstraction framework. Abstract interpretation provides these benefits, but at the cost of added complexity and resistance to mechanized verification. In Section 4.3 we present an alternative perspective: abstract interpretation with *constructive* Galois connections—the topic of this chapter. Constructive Galois connections marry the two worlds presented in this section, providing the simplicity of direct verification, the

benefits of a general abstraction framework, and support for mechanized verification.

To demonstrate both verification perspectives we design a parity analyzer in each style. For example, a parity analysis discovers that 2 has parity `even`, $succ(1)$ has parity `even`, and $n + n$ has parity `even` if n has parity `odd`. Rather than sketch the high-level details of a complete static analyzer, we instead zoom into the low-level details of a tiny fragment: analyzing the successor arithmetic operation $succ(n)$. At this level of detail the differences, advantages and disadvantages of each approach become apparent.

4.2.1 The Direct Approach

Using the direct approach to verification one designs the analyzer, defines what it means for the analyzer to be sound, and then completes a proof of soundness. Each step is done from scratch, and in the simplest way possible.

This approach should be familiar to most readers, and exemplifies how most researchers approach formalizing soundness for static analyzers: first posit the analyzer and soundness framework, then attempt the proof of soundness. One limitation of this approach is that the setup—which gives lots of room for error—isn't known to be correct until after completing the final proof. However, a benefit of this approach is it can easily be mechanized.

Analyzing Successor A parity analysis answers questions like: "what is the parity of $succ(n)$, given that n is even?" To answer these questions, imagine replacing n with the symbol `even`, a stand-in for an arbitrary even number. This hypothetical

expression $succ(\texttt{even})$ is interpreted by defining a successor function over parities, rather than numbers, which we call $succ^\sharp$. This successor operation on parities is designed such that if p is the parity for n, $succ^\sharp(p)$ will be the parity of $succ(n)$:

$$\begin{array}{rcl@{\qquad\qquad}rcl} \mathbb{P} & \coloneqq & \{\texttt{even}, \texttt{odd}\} & succ^\sharp(\texttt{even}) & \coloneqq & \texttt{odd} \\ succ^\sharp & : & \mathbb{P} \to \mathbb{P} & succ^\sharp(\texttt{odd}) & \coloneqq & \texttt{even} \end{array}$$

Soundness The soundness of $succ^\sharp$ is defined using an interpretation for parities, which we notate $[\![p]\!]$:

$$[\![\_]\!] \; : \; \mathbb{P} \to \wp(\mathbb{N}) \qquad\qquad \begin{array}{rcl} [\![\texttt{even}]\!] & \coloneqq & \{n \mid even(n)\} \\ [\![\texttt{odd}]\!] & \coloneqq & \{n \mid odd(n)\} \end{array}$$

Given this interpretation, a parity p is a valid analysis result for a number n if the interpretation for p contains n, that is $n \in [\![p]\!]$. The analyzer $succ^\sharp(p)$ is then sound if, when p is a valid analysis result for some number n, $succ^\sharp(p)$ is a valid analysis result for $succ(n)$:

$$n \in [\![p]\!] \implies succ(n) \in [\![succ^\sharp(p)]\!] \qquad \text{(DA-Snd)}$$

The proof is by case analysis on p; we show the case $p = \texttt{even}$:

$$\begin{array}{lll} n \in [\![\texttt{even}]\!] & & \\ \Leftrightarrow even(n) & \wr \;\; \text{defn. of } [\![\_]\!] & \int \\ \Leftrightarrow odd(succ(n)) & \wr \;\; \text{defn. of } even/odd & \int \\ \Leftrightarrow succ(n) \in [\![\texttt{odd}]\!] & \wr \;\; \text{defn. of } [\![\_]\!] & \int \\ \Leftrightarrow succ(n) \in [\![succ^\sharp(\texttt{even})]\!] & \wr \;\; \text{defn. of } succ^\sharp & \int \end{array}$$

An Even Simpler Setup There is another way to define and prove soundness: use a function which computes the parity of a number in the definition of soundness. This approach is even simpler, and will foreshadow the constructive Galois connection

setup.

$$parity \ : \ \mathbb{N} \to \mathbb{P} \qquad \begin{aligned} parity(0) &\coloneqq \texttt{even} \\ parity(succ(n)) &\coloneqq flip(parity(n)) \end{aligned}$$

where $flip(\texttt{even}) \coloneqq \texttt{odd}$ and $flip(\texttt{odd}) \coloneqq \texttt{even}$. This gives an alternative and equivalent way to relate a number and a parity, due to the following correspondence:

$$n \in [\![p]\!] \iff parity(n) = p \tag{DA-Corr}$$

The soundness of the analyzer is then restated:

$$parity(n) = p \implies parity(succ(n)) = succ^\sharp(p)$$

or by substituting $parity(n) = p$:

$$parity(succ(n)) = succ^\sharp(parity(n)) \tag{DA-Snd*}$$

Both this statement for soundness and its proof are simpler than before. The proof follows directly from the definition of *parity* and the fact that $succ^\sharp$ is identical to $flip$.

The Main Idea Correspondences like (DA-Corr)—between an interpretation for analysis results ($[\![p]\!]$) and a function which computes them ($parity(n)$)—are central to the constructive Galois Connection framework we will describe in Section 4.3. Using correspondences like these, we build a general theory of abstraction that recovers this direct approach to verification, mirrors all of the benefits of abstract interpretation with classical Galois connections, supports mechanized verification, and in some cases simplifies the proof effort. We also observe that many classical

Galois connections used in practice can be ported to this simpler setting.

MECHANIZED VERIFICATION This direct approach to verification is amenable to mechanization using proof assistants like Coq and Agda. These tools are founded on constructive logic in part to support verified program extraction. In constructive logic, functions $f : A \to B$ are computable and often defined inductively to ensure they can be extracted and executed as programs. Analogously, powersets $X : \wp(A)$ are encoded constructively as undecidable predicates $P : A \to \mathit{prop}$ where $x \in X \Leftrightarrow P(x)$.

To mechanize the verification of $\mathit{succ}^\sharp$ we first translate its definition to a constructive setting unmodified. Next we translate $[\![p]\!]$ to a relation $I(p, n)$ defined inductively on n:

$$\frac{}{I(\texttt{even}, 0)} \qquad\qquad \frac{I(p, n)}{I(\mathit{flip}(p), \mathit{succ}(n))}$$

The mechanized proof of (DA-Snd) using I is analogous to the one we sketched, and the mechanized proof of (DA-Snd*) follows directly by computation. The proof term for (DA-Snd*) in both Coq and Agda is simply `refl`, the reflexivity judgment for syntactic equality modulo computation in constructive logic.

WRAPPING UP The two different approaches to verification we present are distinguished by which parts are postulated, and which parts are derived. Using the direct approach, the analyzer ($\mathit{succ}^\sharp$), the interpretation for parities ($[\![p]\!]$) and the definition of soundness (DA-Snd) are postulated up-front. When the soundness setup is correct but the analyzer is wrong, the proof at the end will not go through and the analyzer

must be redesigned. Even worse, when *both* the soundness setup and analyzer are wrong, the proof might actually succeed, giving a false assurance in the soundness of the analyzer. However, the direct approach is attractive because it is simple and supports mechanized verification.

4.2.2 Classical Abstract Interpretation

To verify an analyzer using abstract interpretation with Galois connections, one first designs *abstraction* and *concretization* mappings between sets $\mathbb{N}$ and $\mathbb{P}$. These mappings are used to synthesize an optimal specification for $succ^\sharp$. One then proves that a postulated $succ^\sharp$ meets this synthesized specification, or alternatively derives the definition of $succ^\sharp$ directly from the optimal specification.

In contrast to the direct approach, rather than design the definition of *soundness*, one instead designs the definition of *abstraction* within a structured framework. Soundness is therefore not *designed*, it is *derived* directly from the definition of abstraction. Finally, there is added boilerplate in the abstract interpretation approach, which requires lifting definitions and proofs to powersets.

Abstracting Sets Powersets are introduced in abstraction and concretization functions to support relational mappings, like mapping the symbol `even` to the set of all even numbers. The mappings are therefore between *powersets* $\wp(\mathbb{N})$ and $\wp(\mathbb{P})$. The abstraction and concretization mappings must also satisfy correctness criteria, detailed below, after which they are called a *Galois connection*.

The abstraction mapping from $\wp(\mathbb{N})$ to $\wp(\mathbb{P})$ is notated α, and is defined as

the pointwise lifting of $parity(n)$:

$$\alpha \ : \ \wp(\mathbb{N}) \to \wp(\mathbb{P}) \qquad\qquad \alpha(N) \ \coloneqq \ \{parity(n) \mid n \in N\}$$

The concretization mapping from $\wp(\mathbb{P})$ to $\wp(\mathbb{N})$ is notated γ, and is defined as the flattened pointwise lifting of $[\![p]\!]$:

$$\gamma \ : \ \wp(\mathbb{P}) \to \wp(\mathbb{N}) \qquad\qquad \gamma(P) \ \coloneqq \ \{n \mid p \in P \wedge n \in [\![p]\!]\}$$

The correctness criteria for α and γ is the following correspondence:

$$N \subseteq \gamma(P) \iff \alpha(N) \subseteq P \tag{GC-Corr}$$

The correspondence means that, to relate elements of different sets—in this case $\wp(\mathbb{N})$ and $\wp(\mathbb{P})$—it is equivalent to relate them through either α or γ. Mappings like α and γ which share this correspondence are called Galois connections.

An equivalent formulation of (GC-Corr) is two laws relating compositions of α and γ, called *expansive* and *reductive*:

$$N \subseteq \gamma(\alpha(N)) \tag{GC-Exp}$$

$$\alpha(\gamma(P)) \subseteq P \tag{GC-Red}$$

Property (GC-Red) ensures α is the best abstraction possible w.r.t. γ. For example, a hypothetical definition $\alpha(N) \ \coloneqq \ \{\texttt{even}, \texttt{odd}\}$ is expansive, but not reductive because $\alpha(\gamma(\{\texttt{even}\})) \not\subseteq \{\texttt{even}\}$.

In general, Galois connections are defined for arbitrary *posets* $\langle A, \sqsubseteq^A \rangle$ and $\langle B, \sqsubseteq^B \rangle$. The correspondence (GC-Corr) and its expansive/reductive variants are

generalized in this setting to use partial orders $\sqsubseteq^A$ and $\sqsubseteq^B$ instead of subset ordering. We are omitting monotonicity requirements for α and γ at this point in our presentation, although these requirements are essential in the complete approach.

POWERSET LIFTING The original functions *succ* and $succ^\sharp$ cannot be related through α and γ because they are not functions between powersets. To remedy this they are lifted pointwise:

$$\begin{aligned} \uparrow succ &: \wp(\mathbb{N}) \to \wp(\mathbb{N}) & \uparrow succ(N) &\coloneqq \{succ(n) \mid n \in N\} \\ \uparrow succ^\sharp &: \wp(\mathbb{P}) \to \wp(\mathbb{P}) & \uparrow succ^\sharp(P) &\coloneqq \{succ^\sharp(p) \mid p \in P\} \end{aligned}$$

These lifted operations are called the *concrete interpreter* and *abstract interpreter*, because the former operates over the *concrete domain* $\wp(\mathbb{Z})$ and the latter over the *abstract domain* $\wp(\mathbb{P})$. In the framework of abstract interpretation, static analyzers are just abstract interpreters. Lifting to powersets is necessary to use the abstract interpretation framework, and has the negative effect of adding boilerplate to definitions and proofs of soundness.

SOUNDNESS The definition of soundness for $succ^\sharp$ is synthesized by relating $\uparrow succ^\sharp$ to $\uparrow succ$ composed with α and γ:

$$\alpha(\uparrow succ(\gamma(P))) \subseteq \uparrow succ^\sharp(P) \qquad \text{(GC-Snd)}$$

The left-hand side of the ordering is an optimal specification for any abstraction of $\uparrow succ$ (optimality being a consequence of (GC-Corr)), and the subset ordering says $\uparrow succ^\sharp$ is an over-approximation of this optimal specification. The reason to over-approximate is because the specification is a mathematical description, and the

abstract interpreter is usually an algorithm, and therefore not always able to match the specification precisely. The proof of (GC-Snd) is by case analysis on P. We do not show the proof, rather we demonstrate a proof later in this section which also synthesizes the definition of $succ^\sharp$.

One advantage of the abstract interpretation framework is that it provides a choice between four soundness properties, all of which are generated by α and γ, and equivalent as a consequence of (GC-Corr):

$$\alpha(\uparrow succ(\gamma(P))) \subseteq \uparrow succ^\sharp(P) \qquad \text{(GC-Snd/}\alpha\gamma\text{)}$$

$$\uparrow succ(\gamma(P)) \subseteq \gamma(\uparrow succ^\sharp(P)) \qquad \text{(GC-Snd/}\gamma\gamma\text{)}$$

$$\alpha(\uparrow succ(N)) \subseteq \uparrow succ^\sharp(\alpha(N)) \qquad \text{(GC-Snd/}\alpha\alpha\text{)}$$

$$\uparrow succ(N) \subseteq \gamma(\uparrow succ^\sharp(\alpha(N))) \qquad \text{(GC-Snd/}\gamma\alpha\text{)}$$

Because each soundness property is equivalent, one can choose whichever variant is easiest to prove. The soundness setup (GC-Snd) is the $\alpha\gamma$ rule, however any of the other rules can also be used. For example, one could choose $\alpha\alpha$ or $\gamma\alpha$; in these cases the proof considers four disjoint cases for N: N is empty, N contains only even numbers, N contains only odd numbers, and N contains both even and odd numbers.

Completeness The mappings α and γ also synthesize an *optimality* statement for $\uparrow succ^\sharp$, by stating that it *under*-approximates the optimal specification:

$$\alpha(\uparrow succ(\gamma(P))) \supseteq \uparrow succ^\sharp(P)$$

Because the left-hand-side is an optimal specification, an abstract interpreter will never be strictly more precise. Therefore, optimality is written equivalently using an equality:

$$\alpha(\uparrow succ(\gamma(P))) = \uparrow succ^\sharp(P) \qquad \text{(GC-Opt)}$$

Not all analyzers are optimal, however optimality helps identify those which approximate too much. Consider the analyzer $\uparrow succ^{\sharp\prime}$:

$$\uparrow succ^{\sharp\prime} \ : \ \wp(\mathbb{P}) \to \wp(\mathbb{P}) \qquad\qquad \uparrow succ^{\sharp\prime}(P) \ := \ \{\texttt{even}, \texttt{odd}\}$$

This analyzer reports that $succ(n)$ could have any parity regardless of the parity for n; it's the analyzer that always says "I don't know." This analyzer is perfectly sound but non-optimal because $\uparrow succ^{\sharp\prime}(\{\texttt{even}\}) = \{\texttt{even}, \texttt{odd}\} \neq \alpha(\uparrow succ(\gamma(\{\texttt{even}\})))$.

Just like soundness, four completeness statements are generated by α and γ, however each of these statements are *not* equivalent:

$$\alpha(\uparrow succ(\gamma(P))) = \uparrow succ^\sharp(P) \qquad \text{(GC-Cmp/}\alpha\gamma\text{)}$$

$$\uparrow succ(\gamma(P)) = \gamma(\uparrow succ^\sharp(P)) \qquad \text{(GC-Cmp/}\gamma\gamma\text{)}$$

$$\alpha(\uparrow succ(N)) = \uparrow succ^\sharp(\alpha(N)) \qquad \text{(GC-Cmp/}\alpha\alpha\text{)}$$

$$\uparrow succ(N) = \gamma(\uparrow succ^\sharp(\alpha(N))) \qquad \text{(GC-Cmp/}\gamma\alpha\text{)}$$

Abstract interpreters which satisfy the $\alpha\gamma$ variant are called *optimal* because they lose no more information than necessary, and those which satisfy the $\gamma\alpha$ variant are called *precise* because they lose no information *at all*. The abstract interpreter $succ^\sharp$ is optimal, but not precise because $\gamma(\uparrow succ^\sharp(\alpha(\{1\}))) \neq \uparrow succ(\{1\})$.

To overcome mechanization issues with Galois connections, the state-of-the-art is restricted to use $\gamma\gamma$ rules only for soundness (GC-Snd/$\gamma\gamma$) and completeness (GC-Cmp/$\gamma\gamma$). This is unfortunate for completeness properties because unlike soundness, each completeness variant is not equivalent.

Calculational Derivation of Abstract Interpreters Rather than posit $\uparrow succ^\sharp$ and prove it correct directly, one can instead derive its definition through a calculational process. The process begins with the optimal specification on the left-hand-side of (GC-Opt), and reasons equationally towards the definition of an algorithm. In this way, $\uparrow succ^\sharp$ is not postulated, rather it is derived by calculation, and the result is both sound and optimal by construction.

The derivation is by case analysis on P which has four cases: $\{\}$, $\{\texttt{even}\}$, $\{\texttt{odd}\}$ and $\{\texttt{even}, \texttt{odd}\}$; we show $P = \{\texttt{even}\}$:

$$
\begin{aligned}
&\alpha(\uparrow succ(\gamma(\{\texttt{even}\}))) \\
&= \alpha(\uparrow succ(\{n \mid even(n)\}))\ \wr \quad \text{defn. of } \gamma \quad \int \\
&= \alpha(\{succ(n) \mid even(n)\})\ \wr \quad \text{defn. of } \uparrow succ \quad \int \\
&= \alpha(\{n \mid odd(n)\})\ \wr \quad \text{defn. of } even/odd \quad \int \\
&= \{\texttt{odd}\}\ \wr \quad \text{defn. of } \alpha \quad \int \\
&\triangleq \uparrow succ^\sharp(\{\texttt{even}\})\ \wr \quad \text{defining } \uparrow succ^\sharp \quad \int
\end{aligned}
$$

The derivations for the other cases are analogous, and together they define the implementation of $\uparrow succ^\sharp$.

Deriving analyzers by calculus is attractive because it is systematic, and because it prevents the issue where an analyzer is postulated and discovered to be unsound only after failing to complete its soundness proof. However, this calculational style

of abstract interpretation is not amenable to mechanized verification with program extraction because α is often non-constructive, an issue we describe later in this section.

Added Complexity The abstract interpretation approach requires a Galois connection up-front which necessitates the introduction of powersets $\wp(\mathbb{N})$ and $\wp(\mathbb{P})$. This results in powerset-lifted definitions and adds boilerplate set-theoretic reasoning to the proofs.

This is in contrast to the direct approach which never mentions powersets of parities. Not using powersets results in more understandable soundness criteria, requires no boilerplate set-theoretic reasoning, and results in fewer cases for the proof of soundness. This boilerplate becomes magnified in a mechanized setting where all details must be spelled out to a proof assistant. Furthermore, the simpler proof of (DA-Snd*)—which was immediate from the definition of *parity*—cannot be recovered within the general abstract interpretation framework, rather it must be formulated as a special case. Therefore, in the current state of affairs, one is required to abandon potentially simpler proof techniques in exchange for the benefits of the abstract interpretation framework.

Resistance to Mechanized Verification Despite the beauty and utility of abstract interpretation with Galois connections, advocates of the approach have yet to reconcile their use with advances in mechanized reasoning: *every mechanized verification of an executable abstract interpreter to-date has resisted the use of Galois*

connections, even when initially designed on paper to take advantage of the framework.

The issue in mechanizing Galois connections amounts to a conflict between supporting both classical set-theoretic reasoning and executable static analyzers. Supporting executable analyzers calls for constructive mathematics, which is a problem for α functions because they are often non-constructive, an observation first made by Monniaux [1998]. To work around this limitation, Pichardie [2005] advocates for designing abstract interpreters which are merely inspired by Galois connections, but ultimately avoid their use in verification, which he terms the "γ-only" approach. Successful verification projects such as Verasco adopt this "γ-only" technique [Jourdan et al., 2015, Leroy, 2009], despite the use of Galois connections in designing the original Astrée analyzer [Blanchet et al., 2003].

To understand the foundational issues with Galois connections and α functions, consider verifying the soundness of the parity analyzer using a proof assistant and abstract interpretation. In this setting, the encoding of the Galois connection must support elements of infinite powersets—like the set of all even numbers—as well as executable abstract interpreters which manipulate elements of finite powersets—like $\{\texttt{even}, \texttt{odd}\}$. To support representing infinite sets, the powerset $\wp(\mathbb{N})$ is modeled constructively as a predicate $\mathbb{N} \to \mathit{prop}$. To support defining executable analyzers that manipulate finite sets of parities, the powerset $\wp(\mathbb{P})$ is modeled as an enumeration of its inhabitants, which we call $\mathbb{P}^c$:

$$\mathbb{P}^c \coloneqq \{\texttt{even}, \texttt{odd}, \bot, \top\}$$

where $\bot$ and $\top$ represent $\{\}$ and $\{\texttt{even}, \texttt{odd}\}$. This enables a definition for $\uparrow \mathit{succ}^\sharp$:

$\mathbb{P}^c \to \mathbb{P}^c$ which can be extracted and executed. The consequence of this design is a Galois connection between $\mathbb{N} \to \mathit{prop}$ and $\mathbb{P}^c$; the issue is now α:

$$\alpha \ : \ (\mathbb{N} \to \mathit{prop}) \to \mathbb{P}^c$$

This version of α cannot be defined constructively, as doing would require deciding arbitrary predicates ϕ where $\phi \ : \ \mathbb{N} \to \mathit{prop}$. A hypothetical definition for α would perform case analysis on predicates like $\exists n, \phi(n) \wedge \mathit{even}(n)$ to *compute* an element of $\mathbb{P}^c$, which is not possible for arbitrary ϕ. (The exercise also fails if powersets are modeled with decidable predicates $\phi \ : \ \mathbb{N} \to \mathbb{B}$.) However, γ *can* be defined constructively as a relation (2-ary proposition):

$$\gamma \ : \ \mathbb{P}^c \to (\mathbb{N} \to \mathit{prop})$$

In general, any *theorem* of soundness using Galois connections can be rewritten to use only γ, making use of (GC-Corr); this is the essence of the "γ-only" approach, embodied by the soundness variant (GC-Snd/$\gamma\gamma$). However, this principle does not apply to all *proofs* of soundness using Galois connections, many of which mention α in practice. For example, the γ-only setup does not support calculation in the style advocated by Cousot [1999]. Furthermore, not all *completeness* theorems can be translated to γ-only style, such as (GC-Cmp/$\gamma\alpha$) which is used to show an abstract interpreter is fully precise.

Wrapping Up Abstract interpretation differs from the direct approach in which parts of the design are postulated and which parts are derived. The direct approach

requires postulating the analyzer and definition of soundness. Using abstract interpretation, a Galois connection between sets is postulated instead, and definitions for soundness and completeness are synthesized from the Galois connection. Also, abstract interpretation support deriving the definition of a static analyzer directly from its proof of correctness.

The downside of abstract interpretation is that it requires lifting *succ* and $succ^\sharp$ into powersets, which results in boilerplate set-theoretic reasoning in the proof of soundness. Finally, due to foundational issues, the abstract interpretation framework is not amenable to mechanized verification while also supporting program extraction using constructive logic.

4.3 Constructive Galois Connections

In this section we describe abstract interpretation with constructive Galois connections: a parallel universe of Galois connections analogous to classical ones. The framework enjoys all the benefits of abstract interpretation, but like the direct approach avoids the pitfalls of added complexity and resistance to mechanized verification.

We will describe the framework of constructive Galois connections between sets A and B. When instantiated to $\mathbb{N}$ and $\mathbb{P}$, the framework recovers exactly the direct approach from Section 4.2.1. We will also describe constructive Galois connections in the absence of partial orders, or more specifically, we will assume the discrete partial order: $x \sqsubseteq y \Leftrightarrow x = y$. (Partial orders didn't appear in our demonstration of classical

abstract interpretation, but they are essential to the general theory.) We describe generalizing to partial orders and recovering classical results from constructive ones at the end of this section. The fully general theory of constructive Galois connections is described in Section 4.6 where it is compared side-by-side to classical Galois connections.

Abstracting Sets A constructive Galois connection between sets A and B contains two mappings: the first is called *extraction*, notated η, and the second is called *interpretation*, notated μ:

$$\eta \ : \ A \to B \qquad\qquad \mu \ : \ B \to \wp(A)$$

η and μ are analogous to classical Galois connection mappings α and γ. In the parity analysis described in Section 4.2.1, the extraction function was *parity* and the interpretation function was $[\![\_]\!]$.

Constructive Galois connection mappings η and μ must form a correspondence similar to (GC-Corr):

$$x \in \mu(y) \iff \eta(x) = y \qquad \text{(CGC-Corr)}$$

The intuition behind the correspondence is the same as before: to compare an element x in A to an element y in B, it is equivalent to compare them through either η or μ.

Like classical Galois connections, the correspondence between η and μ is stated equivalently through two composition laws. Extraction functions η which form a constructive Galois connection are also a "best abstraction," analogously to α in the

classical setup:

$$x \in \mu(\eta(x)) \tag{CGC-Ext}$$

$$x \in \mu(y) \implies \eta(x) = y \tag{CGC-Red}$$

ASIDE We use the term *extraction function* and symbol η from Nielson et al. [1999] where η is used to simplify the definition of an abstraction function α. We recover α functions from η in a similar way. However, their treatment of η is a side-note to simplifying the definition of α and nothing more. We take this simple idea much further to realize an entire theory of abstraction around η/μ functions and their correspondences. In this "lowered" theory of η/μ we describe soundness/optimality criteria and calculational derivations analogous to that of α/γ while support mechanized verification, none of which is true of Nielson et al.'s use of η.

INDUCED SPECIFICATIONS Four equivalent soundness criteria are generated by η and μ just like in the classical framework. Each soundness statement uses η and μ in a different but equivalent way (assuming (CGC-Corr)). For a concrete $f : A \to A$ and abstract $f^\sharp : B \to B$, $f^\sharp$ is sound *iff* any of the following properties hold:

$$x \in \mu(y) \wedge y' = \eta(f(x)) \implies y' = f^\sharp(y) \tag{CGC-Snd/$\eta\mu$}$$

$$x \in \mu(y) \wedge x' = f(x) \implies x' \in \mu(f^\sharp(y)) \tag{CGC-Snd/$\mu\mu$}$$

$$y = \eta(f(x)) \implies y = f^\sharp(\eta(x)) \tag{CGC-Snd/$\eta\eta$}$$

$$x' = f(x) \implies x' \in \mu(f^\sharp(\eta(x))) \tag{CGC-Snd/$\mu\eta$}$$

In the direct approach to verifying an example parity analysis described in Section 4.2.1, the first soundness property (DA-Snd) is generated by the $\mu\mu$ variant, and the second soundness property (DA-Snd*) which enjoyed a simpler proof is generated by the $\eta\eta$ variant. We write these soundness rules in a slightly strange way so we can write their completeness analogs simply by replacing $\Rightarrow$ with $\Leftrightarrow$. The origin of these rules comes from an adjunction framework, which we discuss in Section 4.6.

The mappings η and μ also generate four completeness criteria which, like classical Galois connections, are not equivalent:

$$x \in \mu(y) \wedge y' = \eta(f(x)) \iff y' = f^\sharp(y) \qquad \text{(CGC-Cmp/}\eta\mu\text{)}$$

$$x \in \mu(y) \wedge x' = f(x) \iff x' \in \mu(f^\sharp(y)) \qquad \text{(CGC-Cmp/}\mu\mu\text{)}$$

$$y = \eta(f(x)) \iff y = f^\sharp(\eta(x)) \qquad \text{(CGC-Cmp/}\eta\eta\text{)}$$

$$x' = f(x) \iff x' \in \mu(f^\sharp(\eta(x))) \qquad \text{(CGC-Cmp/}\mu\eta\text{)}$$

Inspired by classical Galois connections, we call abstract interpreters $f^\sharp$ which satisfy the $\eta\mu$ variant *optimal* and those which satisfy the $\mu\eta$ variant *precise*.

The above soundness and completeness rules are stated for concrete and abstraction *functions* $f : A \to A$ and $f^\sharp : B \to B$. However, they generalize easily to *relations* $R : \wp(A \times A)$ and *predicate transformers* $F : \wp(A) \to \wp(A)$ (*i.e.* collecting semantics) through the adjunction framework discussed in Section 4.6. The case studies in Sections 4.4 and 4.5 describe abstract interpreters over concrete relations and their soundness conditions.

Calculational Derivation of Abstract Interpreters The constructive Galois connection framework also supports deriving abstract interpreters through calculation, analogously to the calculation we demonstrated in Section 4.2.2. To support calculational reasoning, the four logical soundness criteria are rewritten into statements about subsumption between powerset elements:

$$\{\eta(f(x)) \mid x \in \mu(y)\} \subseteq \{f^\sharp(y)\} \qquad \text{(CGC-Snd/}\eta\mu^*\text{)}$$

$$\{f(x) \mid x \in \mu(y)\} \subseteq \mu(f^\sharp(y)) \qquad \text{(CGC-Snd/}\mu\mu^*\text{)}$$

$$\{\eta(f(x))\} \subseteq \{f^\sharp(\eta(x))\} \qquad \text{(CGC-Snd/}\eta\eta^*\text{)}$$

$$\{f(x)\} \subseteq \mu(f^\sharp(\eta(x))) \qquad \text{(CGC-Snd/}\mu\eta^*\text{)}$$

The completeness analog to the four rules replaces set subsumption with equality. Using the $\eta\mu^*$ completeness rule, one calculates towards a definition for $f^\sharp$ starting from the left-hand-side, which is the optimal specification for abstract interpreters of f.

To demonstrate calculation using constructive Galois connections, we show the derivation of $succ^\sharp$ from its induced specification, the result of which is sound and optimal (because each step is $=$ in addition to $\subseteq$) by construction; we show

$p = \texttt{even}$:

$$
\begin{aligned}
&\{parity(succ(n)) \mid n \in [\![\texttt{even}]\!]\} \\
&= \{parity(succ(n)) \mid even(n)\} && \wr \text{ defn. of } [\![\_]\!] \int \\
&= \{flip(parity(n)) \mid even(n)\} && \wr \text{ defn. of } parity \int \\
&= \{flip(\texttt{even})\} && \wr \text{ Eq. DA-Corr } \int \\
&= \{\texttt{odd}\} && \wr \text{ defn. of } flip \int \\
&\triangleq \{succ^\sharp(\texttt{even})\} && \wr \text{ defining } succ^\sharp \int
\end{aligned}
$$

We will show another perspective on this calculation later in this section, where the derivation of $succ^\sharp$ is not only sound and optimal by construction, but computable by construction as well.

Mechanized Verification In addition to the benefits of a general abstraction framework, constructive Galois connections are amenable to mechanization in a way that classical Galois connections are not. In our Agda library and case studies we mechanize constructive Galois connections in full generality, as well as proofs that use both mapping functions, such as calculational derivations.

As we discussed in Sections 4.2.1 and 4.2.2, the constructive encoding for infinite powersets $\wp(A)$ is $A \to prop$. This results in the following types for η and μ when encoded constructively:

$$\eta \;:\; \mathbb{N} \to \mathbb{P} \qquad\qquad \mu \;:\; \mathbb{P} \to \mathbb{N} \to prop$$

In constructive logic, the arrow type $\mathbb{N} \to \mathbb{P}$ classifies computable functions, and the arrow type $\mathbb{P} \to \mathbb{N} \to prop$ classifies undecidable relations. (CGC-Corr) is then

mechanized without issue:

$$\mu(p, n) \iff \eta(n) = p$$

See the mechanization details in Section 4.2.1 for how η and μ are defined constructively for the example parity analysis.

Wrapping Up Constructive Galois connections are a general abstraction framework similar to classical Galois connections. At the heart of the constructive Galois connection framework is a correspondence (CGC-Corr) analogous to its classical counterpart. From this correspondence, soundness and completeness criteria are synthesized for abstract interpreters. Constructive Galois connections also support calculational derivations of abstract interpreters which and sound and optimal by construction. In addition to these benefits of a general abstraction framework, constructive Galois connections are amenable to mechanized verification. Both extraction (η) and interpretation (μ) can be mechanized effectively, as well as proofs of soundness, completeness, and calculational derivations.

4.3.1 Partial Orders and Monotonicity

The full theory of constructive Galois connections generalizes to posets $\langle A, \sqsubseteq^A \rangle$ and $\langle B, \sqsubseteq^B \rangle$ by making the following changes:

- Powersets must be downward-closed, that is for $X : \wp(A)$:

$$x \in X \wedge x' \sqsubseteq x \implies x' \in X \qquad \text{(PowerMon)}$$

Singleton sets $\{x\}$ are reinterpreted to mean $\{x' \mid x' \sqsubseteq x\}$. For mechanization, this means $\wp(A)$ is encoded as an *antitonic* function, notated with a down-right arrow $A \searrow \mathit{prop}$, where the partial ordering on *prop* is by implication.

- Functions must be monotonic, that is for $f : A \to A$:

$$x \sqsubseteq x' \implies f(x) \sqsubseteq f(x') \qquad \text{(FunMon)}$$

We notate monotonic functions $f : A \nearrow A$. Monotonicity is required for mappings η and μ, and concrete and abstract interpreters f and $f^\sharp$.

- The constructive Galois connection correspondence is generalized to partial orders in place of equality, that is for η and μ:

$$x \in \mu(y) \iff \eta(x) \sqsubseteq y \qquad \text{(CGP-Corr)}$$

or alternatively, by generalizing the reductive property:

$$x \in \mu(y) \implies \eta(x) \sqsubseteq y \qquad \text{(CGP-Red)}$$

- Soundness criteria are also generalized to partial orders:

$$x \in \mu(y) \wedge y' \sqsubseteq \eta(f(x)) \implies y' \sqsubseteq f^\sharp(y) \qquad \text{(CGP-Snd/}\eta\mu\text{)}$$

$$x \in \mu(y) \wedge x' \sqsubseteq f(x) \implies x' \in \mu(f^\sharp(y)) \qquad \text{(CGP-Snd/}\mu\mu\text{)}$$

$$y \sqsubseteq \eta(f(x)) \implies y \sqsubseteq f^\sharp(\eta(x)) \qquad \text{(CGP-Snd/}\eta\eta\text{)}$$

$$x' \sqsubseteq f(x) \implies x' \in \mu(f^\sharp(\eta(x))) \qquad \text{(CGP-Snd/}\mu\eta\text{)}$$

We were careful to write the equalities in Section 4.3 in the right order so this

change is just swappping $=$ for $\sqsubseteq$. Completeness criteria are identical with $\Leftrightarrow$ in place of $\Rightarrow$.

To demonstrate when partial orders and monotonicity are necessary, consider designing a parity analyzer for the *max* operator:

$$max^\sharp \ : \ \mathbb{P} \times \mathbb{P} \to \mathbb{P} \qquad \begin{array}{rcl@{\qquad}rcl} max^\sharp(\texttt{even}, \texttt{even}) & \coloneqq & \texttt{even} & max^\sharp(\texttt{even}, \texttt{odd}) & \coloneqq & ? \\ max^\sharp(\texttt{odd}, \texttt{odd}) & \coloneqq & \texttt{odd} & max^\sharp(\texttt{odd}, \texttt{even}) & \coloneqq & ? \end{array}$$

The last two cases for $max^\sharp$ cannot be defined because the maximum of an even and odd number could be either even or odd, and there is no representative for "any number" in $\mathbb{P}$. To remedy this, we add ANY to the set of parities: $\mathbb{P}^+ \coloneqq \mathbb{P} \cup \{\text{ANY}\}$; the new element ANY is interpreted: $[\![\text{ANY}]\!] \coloneqq \{n \mid n \in \mathbb{N}\}$; the partial order on $\mathbb{P}^+$ becomes: $\texttt{even}, \texttt{odd} \sqsubseteq \text{ANY}$; and the correspondence continues to hold using this partial order: $n \in [\![p^+]\!] \iff \mathit{parity}(n) \sqsubseteq p^+$. $max^\sharp$ is then defined using the abstraction $\mathbb{P}^+$ and proven sound and optimal following the abstract interpretation paradigm.

4.3.2 Relationship to Classical Galois Connections

We clarify the relationship between constructive and classical Galois connections in three ways:

- Any constructive Galois connection can be lifted to obtain an equivalent classical Galois connection, and likewise for soundness and completeness proofs.
- Any classical Galois connection which can be recovered by a constructive one contains no additional expressive power, rendering it an equivalent theory with

added boilerplate reasoning.

- Not all classical Galois connections can be recovered by constructive ones.

From these relationships we conclude that one benefits from using constructive Galois connections whenever possible, classical Galois connections when no constructive one exists, and both theories together as needed. We make these claims precise in Section 4.6, and explore the subtleties of their relationship in detail in sections 4.8, 4.9 and 4.10.

A classical Galois connection is recovered from a constructive one through the following lifting:

$$\begin{array}{lll} \alpha & : & \wp(A) \to \wp(B) \qquad \alpha(X) \coloneqq \{\eta(x) \mid x \in X\} \\ \gamma & : & \wp(B) \to \wp(A) \qquad \gamma(Y) \coloneqq \{x \mid y \in Y \wedge x \in \mu(y)\} \end{array}$$

When a classical Galois connection can be written in this form for some η and μ, then one can use the simpler setting of abstract interpretation with constructive Galois connections without any loss of generality. We also observe that many classical Galois connections in practice can be written in this form, and therefore can be mechanized effectively using constructive Galois connections. The case studies in presented in Sections 4.4 and 4.5 are two such cases, although the original authors of those works did not initially write their classical Galois connections in this explicitly lifted form.

An example of a classical Galois connection which is not recovered by lifting is the Independent Attributes (IA) abstraction, which abstracts relations $R : \wp(A \times B)$

with their component-wise splitting $\langle R_l, R_r \rangle \ : \ \wp(A) \times \wp(B)$:

$$\begin{aligned} \alpha \ &: \ \wp(A \times B) \to \wp(A) \times \wp(B) \\ \gamma \ &: \ \wp(A) \times \wp(B) \to \wp(A \times B) \end{aligned}$$

$$\begin{aligned} \alpha(R) \ &\coloneqq \ \langle \{x \mid \exists y. \langle x, y \rangle \in R\}, \{y \mid \exists x. \langle x, y \rangle \in R\} \rangle \\ \gamma(R_l, R_r) \ &\coloneqq \ \{\langle x, y \rangle \mid x \in R_l, y \in R_r\} \end{aligned}$$

This Galois connection *is* amenable to mechanized verification. In a constructive setting, α and γ are maps between $A \times B \to \mathit{prop}$ and $(A \to \mathit{prop}) \times (B \to \mathit{prop})$, and can be defined directly using logical connectives $\exists$ and $\wedge$:

$$\begin{aligned} \alpha(R) \ &\coloneqq \ \langle \lambda x. \exists y. R(x, y), \lambda y. \exists x. R(x, y) \rangle \\ \gamma(R_l, R_r) \ &\coloneqq \ \lambda \langle x, y \rangle . R_l(x) \wedge R_r(y) \end{aligned}$$

IA can be mechanized effectively because the Galois connection consists of mappings between specifications, and the foundational issue of constructing values from specifications does not appear. IA is not a constructive Galois connection because there is no pure function μ underlying the abstraction function α.

Because constructive Galois connections can be lifted to classical ones, a constructive Galois connection can interact directly with IA through its lifting, even in a mechanized setting. However, once a constructive Galois connection is lifted it loses its computational properties and cannot be extracted and executed. In practice, IA is used to weaken ($\sqsubseteq$) an induced optimal specification after which the calculated interpreter is shown to be optimal ($=$) up-to-IA. IA never appears in the final calculated interpreter, so not having a constructive Galois connection formulation poses no issue. We explore how a constructive Galois connection derivation interacts with IA in detail in sections 4.8 and 4.9.

4.3.3 The "Specification Effect"

The machinery of constructive Galois connections follow a *monadic effect* discipline, where the effect type is the classical powerset $\wp(\_)$; we call this a *specification effect*. First we will describe the monadic structure of powersets $\wp(\_)$ and what we mean by "specification effect." Then we will recast the theory of constructive Galois connections in this monadic style, giving insights into why the theory supports mechanized verification, and foreshadowing key fragments of the metatheory we develop in Section 4.6.

The monadic structure of classical powersets is standard, and is analogous to the nondeterminism monad familiar to Haskell programmers. However, the model $\wp(A) \coloneqq A \to \mathit{prop}$ is the uncomputable nondeterminism monad and mirrors the use of set-comprehensions on paper to describe uncomputable sets (specifications), rather than the use of monad comprehensions in Haskell to describe computable sets (constructed values).

We generalize $\wp(\_)$ to a *monotonic* monad, similarly to how we generalized powersets to posets in Section 4.3.1. This results in monotonic versions of monad operators *ret* and *bind*:

$$\begin{array}{rcl@{\qquad\qquad}rcl} \mathit{ret} & : & A \nearrow \wp(A) & \mathit{bind} & : & \wp(A) \times (A \nearrow \wp(B)) \nearrow \wp(B) \\ \mathit{ret}(x) & \coloneqq & \{x' \mid x' \sqsubseteq x\} & \mathit{bind}(X, f) & \coloneqq & \{y \mid x \in X \wedge y \in f(x)\} \end{array}$$

We adopt Moggi's notation [1989] for monadic extension where $\mathit{bind}(X, f)$ is written $f^*(X)$, or just f^* for $\lambda X.f^*(X)$.

We call the powerset type $\wp(A)$ a specification effect because it has monadic

structure, supports encoding arbitrary properties over values in A, and cannot be "escaped from" in constructive logic, similar to the IO monad in Haskell. In classical mathematics, there is an isomorphism between singleton powersets $\wp^1(A)$ and the set A. However, no such constructive mapping exists for $\wp^1(A) \to A$. Such a function would decide arbitrary predicates in $A \to \mathit{prop}$ to *compute* the A inside the singleton set. This observation, that you can program inside $\wp(\_)$ monadically in constructive logic, but you can't escape the monad, is why we call it a specification effect.

Given the monadic structure for powersets, and the intuition that they encode a specification effect in constructive logic, we can recast the theory of constructive Galois connections using monadic operators. To do this we define a helper operator which injects "pure" functions into the "effectful" function space:

$$\mathit{pure} \;:\; (A \nrightarrow B) \nrightarrow (A \nrightarrow \wp(B)) \qquad\qquad \mathit{pure}(f)(x) \;\coloneqq\; \mathit{ret}(f(x))$$

We then rewrite (CGC-Corr) using ret and pure:

$$\mathit{ret}(x) \subseteq \mu(y) \iff \mathit{pure}(\eta)(x) \subseteq \mathit{ret}(y) \qquad \text{(CGM-Corr)}$$

and we rewrite the expansive and reductive variant of the correspondence using ret, *bind* (notated f^*) and pure:

$$\mathit{ret}(x) \subseteq \mu^*(\mathit{pure}(\eta)(x)) \qquad \text{(CGM-Exp)}$$

$$\mathit{pure}(\eta)^*(\mu(y)) \subseteq \mathit{ret}(y) \qquad \text{(CGM-Red)}$$

The four soundness and completeness conditions can also be written in monadic

style; we show the $\eta\mu$ soundness property here:

$$pure(\eta)^*(pure(f)^*(\mu(y))) \subseteq pure(f^\sharp)(y) \qquad \text{(CGM-Snd)}$$

The left-hand-side of the ordering is the optimal specification for $f^\sharp$, just like (CGC-Snd/$\eta\mu$) but using monadic operators. The right-hand-side of the ordering is $f^\sharp$ lifted to the monadic function space. The constructive calculation of $succ^\sharp$ we showed earlier in this section is a calculation of this form. The specification on the left has type $\wp(\mathbb{P})$, and it *has effects*, meaning it uses classical reasoning and can't be executed. The abstract interpreter on the right also has type $\wp(\mathbb{P})$, but it *has no effects*, meaning it *can* be extracted and executed. The calculated abstract interpreter is thus not only sound and optimal by construction, *it is computable by construction.*

Constructive Galois connections are empowering because they treat specification like an effect, which optimal specifications *ought to have*, and which computable abstract interpreters *ought not to have*. Using a monadic effect discipline we support calculations which start with a specification effect, and where the "effect" is eliminated through the process of calculation. The monad laws are crucial in canceling uses of *ret* with *bind* to arrive at a final pure computation. For example, the first step in a derivation for (CGM-Snd) can immediately simplify using monad laws to:

$$pure(\eta \circ f)^*(\mu(y)) \subseteq pure(f^\sharp)(y)$$

4.4 Case Study 1: Calculational AI

In this section we apply constructive Galois connections to the *Calculational Design of a Generic Abstract Interpreter* from Cousot's monograph [1999]. To our knowledge, *we achieve the first mechanically verified abstract interpreter derived by calculus.*

The key challenge in mechanizing the interpreter is supporting both abstraction (α) and concretization (γ) mappings, which are required by the calculational approach. Classical Galois connections do not support mechanization of α without the use of axioms, and these required axioms block computation, preventing the extraction of verified algorithms.

To verify Cousot's generic abstract interpreter we use constructive Galois connections, which we describe in Section 4.3 and formalize in Section 4.6. Using constructive Galois connections we encode extraction (η) and interpretation (μ) mappings as constructive analogs to α and γ, calculate an abstract interpreter for an imperative programming language which is sound and computable by construction, and recover the original classical Galois connection results through a systematic lifting.

First we describe the setup for the analyzer: the abstract syntax, the concrete semantics, and the constructive Galois connections involved. Following the abstract interpretation paradigm with constructive Galois connections we design abstract interpreters for denotation functions and semantics relations. We show a fragment of our Agda mechanization which closely mirrors the pencil-and-paper proof, as well as Cousot's original derivation.

$$
\begin{array}{rrcll}
i \in & \mathbb{Z} & ::= & \{\ldots,-1,0,1,\ldots\} & \text{integers} \\
b \in & \mathbb{B} & ::= & \{true, false\} & \text{booleans} \\
x \in & \texttt{var} & ::= & \ldots & \text{variables} \\
\oplus \in & \texttt{aop} & ::= & + \mid - \mid \times \mid / & \text{arithmetic op.} \\
\circledcirc \in & \texttt{cmp} & ::= & < \mid = & \text{comparison op.} \\
\bigtriangledown \in & \texttt{bop} & ::= & \vee \mid \wedge & \text{boolean op.} \\
ae \in & \texttt{aexp} & ::= & i \mid x \mid \texttt{rand} \mid ae \oplus ae & \text{arithmetic exp.} \\
be \in & \texttt{bexp} & ::= & b \mid ae \circledcirc ae \mid be \bigtriangledown be & \text{boolean exp.} \\
ce \in & \texttt{cexp} & ::= & \texttt{skip} \mid ce\ ;\ ce & \text{skip \& sequence exp.} \\
 & & \mid & x := ae & \text{assignment exp.} \\
 & & \mid & \texttt{if}\ be\ \texttt{then}\ ce\ \texttt{else}\ ce & \text{conditional exp.} \\
 & & \mid & \texttt{while}\ be\ \texttt{do}\ ce & \text{while loop exp.}
\end{array}
$$

Figure 4.1: Case Study 1: `WHILE` Abstract Syntax

4.4.1 Concrete Semantics

The `WHILE` language is an imperative programming language with arithmetic expressions, variable assignment and while-loops. We show the syntax for this language in Figure 4.1. `WHILE` syntactically distinguished arithmetic, boolean and command expressions. `rand` is an arithmetic expression which can evaluate to any integer. Syntactic categories $\oplus$, $\circledcirc$ and $\bigtriangledown$ range over arithmetic, comparison and boolean operators, and are introduced to simplify the presentation. The `WHILE` language is taken from Cousot's monograph [Cousot, 1999].

The concrete semantics of `WHILE` is sketched without full definition in Figure 4.2.

Denotation functions $[\![\_]\!]^a$, $[\![\_]\!]^c$ and $[\![\_]\!]^b$ give semantics to arithmetic, conditional and boolean operators. The semantics of compound syntactic expressions are given operationally with relations $\Downarrow^a$, $\Downarrow^b$ and $\mapsto^c$. Relational semantics are given for arithmetic and boolean expressions due to the nondeterminism of `rand` and, for command expressions due to the nontermination of `while`. These semantics serve as the starting point for designing an abstract interpreter.

4.4.2 Abstract Semantics with Constructive GCs

Using abstract interpretation with constructive Galois connections, we design an abstract semantics for `WHILE` in the following steps:

1. An abstraction for each set $\mathbb{Z}$, $\mathbb{B}$ and `env`.
2. An abstraction for each denotation function $[\![\_]\!]^a$, $[\![\_]\!]^c$ and $[\![\_]\!]^b$.
3. An abstraction for each semantics relation $\Downarrow^a$, $\Downarrow^b$ and $\mapsto^c$.

Each abstract set forms a constructive Galois connection with its concrete counterpart. Soundness criteria is synthesized for abstract functions and relations using constructive Galois connection mappings. Finally, we verify and calculate abstract interpreters from these specifications which are sound and computable by construction. We describe the details of this process only for integers and environments (the sets $\mathbb{Z}$ and `env`), arithmetic operators (the denotation function $[\![\_]\!]^a$), and arithmetic expressions (the semantics relation $\Downarrow^a$). See the Agda development accompanying this chapter for the full mechanization of `WHILE`, and sections 4.8, 4.9, and 4.10

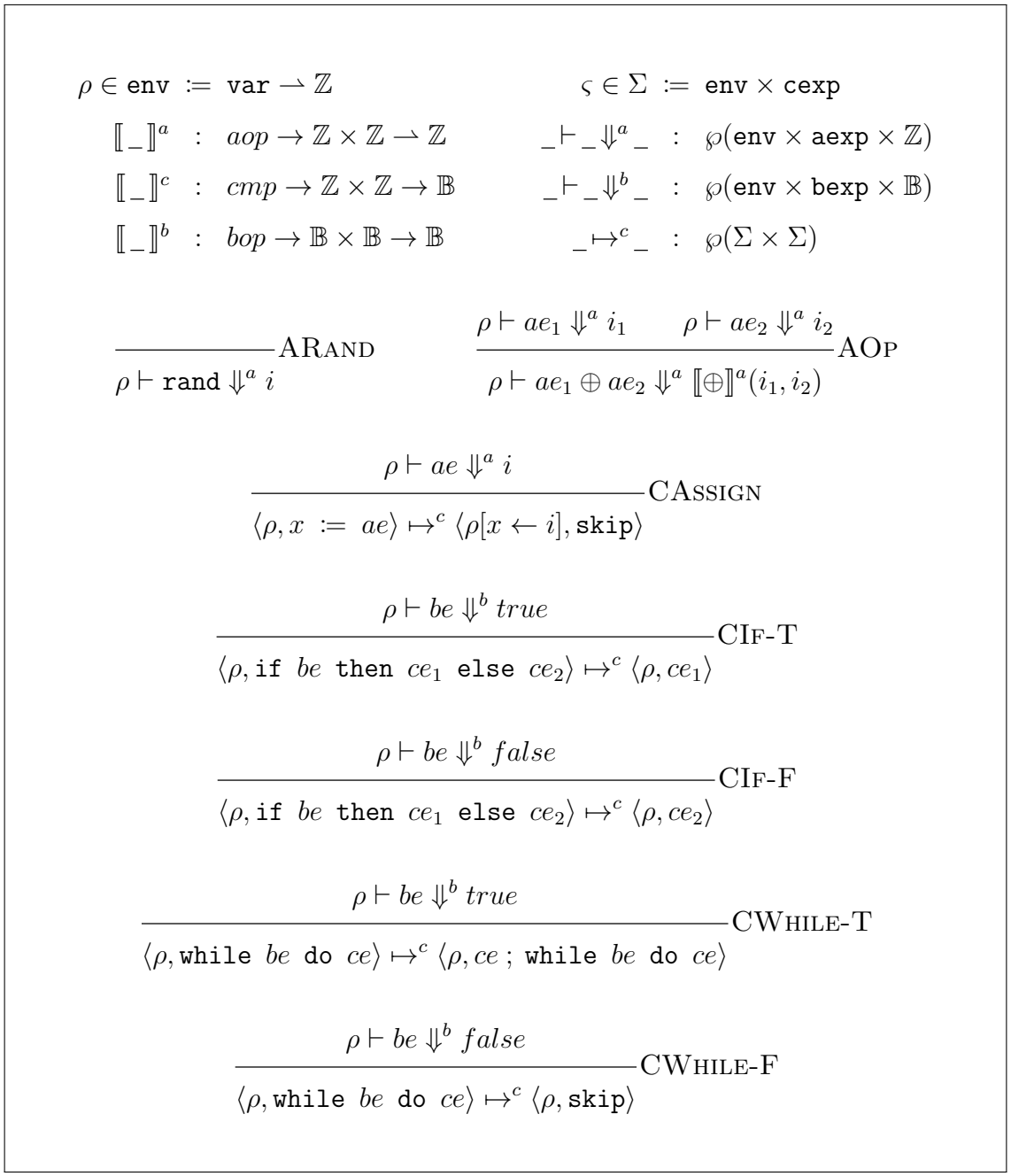


Figure 4.2: Case Study 1: `WHILE` Concrete Semantics

for a detailed account of binary arithmetic operators and conditional command expressions.

Abstracting Integers We design a simple sign abstraction for integers, although more powerful abstractions are certainly possible [Cousot, 1999, Miné, 2006]. The final abstract interpreter for `WHILE` is parameterized by any abstraction for integers, meaning another abstraction can be plugged in without added proof effort.

The sign abstraction begins with three representative elements: `neg`, `zer` and `pos`, representing negative integers, the integer 0, and positive integers. To support representing integers which could be negative or 0, negative or positive, or 0 or positive, etc. we design a set which is complete w.r.t these logical disjunctions:

$$i^\sharp \in \mathbb{Z}^\sharp \coloneqq \{\texttt{none}, \texttt{neg}, \texttt{zer}, \texttt{pos}, \texttt{negz}, \texttt{nzer}, \texttt{posz}, \texttt{any}\}$$

$\mathbb{Z}^\sharp$ is given meaning through an interpretation function μ^z, the analog of a γ from the classical Galois connection framework:

$$\mu^z : \mathbb{Z}^\sharp \nearrow \wp(\mathbb{Z})$$

$$\begin{aligned}
\mu^z(\texttt{none}) &\coloneqq \{\} & \mu^z(\texttt{negz}) &\coloneqq \{i \mid i \leq 0\} \\
\mu^z(\texttt{neg}) &\coloneqq \{i \mid i < 0\} & \mu^z(\texttt{nzer}) &\coloneqq \{i \mid i \neq 0\} \\
\mu^z(\texttt{zer}) &\coloneqq \{0\} & \mu^z(\texttt{posz}) &\coloneqq \{i \mid i \geq 0\} \\
\mu^z(\texttt{pos}) &\coloneqq \{i \mid i > 0\} & \mu^z(\texttt{any}) &\coloneqq \{i \mid i \in \mathbb{Z}\}
\end{aligned}$$

The partial ordering on abstract integers coincides with subset ordering under μ^z, that is, $i_1^\sharp \sqsubseteq^z i_2^\sharp \iff \mu^z(i_1^\sharp) \subseteq \mu^z(i_2^\sharp)$:

$$\texttt{none} \sqsubseteq^z i^\sharp \sqsubseteq^z \texttt{any}$$

$$\begin{aligned}
\texttt{neg} &\sqsubseteq^z \texttt{negz}, \texttt{nzer} \\
\texttt{zer} &\sqsubseteq^z \texttt{negz}, \texttt{posz} \\
\texttt{pos} &\sqsubseteq^z \texttt{nzer}, \texttt{posz}
\end{aligned}$$

To be a constructive Galois connection, μ^z forms a correspondence with a best abstraction function η^z:

$$\eta^z \ : \ \mathbb{Z} \to \mathbb{Z}^\sharp \qquad\qquad \eta^z(n) \ \coloneqq \begin{cases} \texttt{neg} & \mathit{if} \quad i < 0 \\ \texttt{zer} & \mathit{if} \quad i = 0 \\ \texttt{pos} & \mathit{if} \quad i > 0 \end{cases}$$

and we prove the constructive Galois connection correspondence:

$$i \in \mu^z(i^\sharp) \iff \eta^z(i) \sqsubseteq^z i^\sharp$$

The Classical Design To contrast with Cousot's original design using classical abstract interpretation, the key difference is the abstraction function. The abstraction function using classical Galois connections is recovered through a lifting of our η^z:

$$\alpha^z \ : \ \wp(\mathbb{Z}) \nearrow \mathbb{Z}^\sharp \qquad\qquad \alpha^z(I) \ \coloneqq \bigsqcup_{i \in I} \eta^z(i)$$

Abstraction functions of this form—$\wp(B) \nearrow A$, for some concrete set A and abstract set B—are representative of most Galois connections used in the literature for static analyzers. However, these abstraction functions are precisely the part of classical Galois connections which inhibit mechanized verification. The extraction function η^z does not manipulate powersets, does not inhibit mechanized verification, and recovers the original non-constructive α^z through this standard lifting.

Abstracting Environments An abstract environment maps variables to abstract integers rather than concrete integers.

$$\rho^\sharp \in \mathtt{env}^\sharp \coloneqq \mathtt{var} \to \mathbb{Z}^\sharp$$

$\mathtt{env}^\sharp$ is given meaning through an interpretation function μ^r:

$$\mu^r \ : \ \mathtt{env}^\sharp \nearrow \wp(\mathtt{env}) \qquad\qquad \mu^r(\rho^\sharp) \coloneqq \{\rho \mid \forall x.\rho(x) \in \mu^z(\rho^\sharp(x))\}$$

An abstract environment represents concrete environments that agree pointwise with some represented integer in the codomain.

The order on abstract environments is the standard pointwise ordering and coincides with subset ordering under μ^r, that is, $\rho_1^\sharp \sqsubseteq^r \rho_2^\sharp \iff \mu^r(\rho_1^\sharp) \subseteq \mu^r(\rho_2^\sharp)$:

$$\rho_1^\sharp \sqsubseteq^r \rho_2 \coloneqq \forall x.\rho_1^\sharp(x) \sqsubseteq^z \rho_2^\sharp(x)$$

To form a constructive Galois connection, μ^r forms a correspondence with a best abstraction function η^r:

$$\eta^r \ : \ \mathtt{env} \to \mathtt{env}^\sharp \qquad\qquad \eta^r(\rho) \coloneqq \lambda x.\eta^z(\rho(x))$$

and we prove the constructive Galois connection correspondence:

$$\rho \in \mu^r(\rho^\sharp) \iff \eta^r(\rho) \sqsubseteq^r \rho^\sharp$$

The Classical Design To contrast with Cousot's original design using classical abstract interpretation, the key difference is again the abstraction function.

The abstraction function using classical Galois connections is:

$$\alpha^r \;:\; \wp(\texttt{env}) \nearrow \texttt{env}^\sharp \qquad\qquad \alpha^r(R) \;\coloneqq\; \lambda x.\alpha^z(\{\rho(x) \mid \rho \in R\})$$

which is also not amenable to mechanized verification.

Abstracting Functions After designing constructive Galois connections for $\mathbb{Z}$ and $\texttt{env}$ we define what it means for $[\![\_]\!]^{a\sharp}$, some abstract denotation for arithmetic operators, to be a sound abstraction of $[\![\_]\!]^a$, its concrete counterpart. This is done through a specification induced by mappings η and μ, analogously to how specifications are induced using classical Galois connections.

The specification which encodes soundness and optimality for $[\![\_]\!]^{a\sharp}$ is generated using the constructive Galois connection for $\mathbb{Z}$:

$$\langle i_1, i_2\rangle \in \mu^{z\times z}(i_1^\sharp, i_2^\sharp) \wedge \langle i_1^{\sharp\prime}, i_2^{\sharp\prime}\rangle \sqsubseteq \eta^z([\![ae]\!]^a(i_1, i_2)) \Leftrightarrow \langle i_1^{\sharp\prime}, i_2^{\sharp\prime}\rangle \sqsubseteq [\![ae]\!]^{a\sharp}(i_1^\sharp, i_2^\sharp)$$

(See (CGC-Cmp/$\eta\mu$) in Section 4.3 for the origin of this equation.) For $[\![\_]\!]^{a\sharp}$, we postulate its definition and verify its correctness post-facto using the above property, although we omit the proof details here. The definition of $[\![\_]\!]^{a\sharp}$ is standard, and returns $\texttt{none}$ in the case of division by zero. We show only the definition of $+$ here:

$$[\![\_]\!]^{a\sharp} \;:\; \texttt{aexp} \to \mathbb{Z}^\sharp \times \mathbb{Z}^\sharp \nearrow \mathbb{Z}^\sharp$$

$$[\![+]\!]^{a\sharp}(i_1^\sharp, i_2^\sharp) \;\coloneqq\; \bigsqcup \begin{cases} \texttt{pos} & \textit{if} \quad \texttt{pos} \sqsubseteq^z i_1^\sharp \vee \texttt{pos} \sqsubseteq^z i_2^\sharp \\ \texttt{neg} & \textit{if} \quad \texttt{neg} \sqsubseteq^z i_1^\sharp \vee \texttt{neg} \sqsubseteq^z i_2^\sharp \\ \texttt{zer} & \textit{if} \quad \texttt{zer} \sqsubseteq^z i_1^\sharp \wedge \texttt{zer} \sqsubseteq^z i_2^\sharp \\ \texttt{zer} & \textit{if} \quad \texttt{pos} \sqsubseteq^z i_1^\sharp \wedge \texttt{neg} \sqsubseteq^z i_2^\sharp \\ \texttt{zer} & \textit{if} \quad \texttt{neg} \sqsubseteq^z i_1^\sharp \wedge \texttt{pos} \sqsubseteq^z i_2^\sharp \end{cases}$$

The Classical Design To contrast with Cousot's original design using classical abstract interpretation, the key difference is that we avoid powerset liftings all-together. Using classical Galois connections, the concrete denotation function must be lifted to powersets:

$$[\![\_]\!]^a_\wp \ : \ \texttt{aexp} \to \wp(\mathbb{Z} \times \mathbb{Z}) \to \wp(\mathbb{Z}) \qquad [\![ae]\!]^a_\wp(II) \ \coloneqq \ \{[\![ae]\!]^a(i_1, i_2) \mid \langle i_1, i_2 \rangle \in II\}$$

and then $[\![\_]\!]^{a\sharp}$ is proven correct w.r.t. this lifting using α^z and γ^z:

$$\alpha^z([\![ae]\!]^a_\wp(\gamma^z(i^\sharp_1, i^\sharp_2))) = [\![ae]\!]^{a\sharp}(i^\sharp_1, i^\sharp_2)$$

This property cannot be mechanized without axioms because α^z is non-constructive. Furthermore, the proof involves additional powerset boilerplate reasoning, which is not present in our mechanization of correctness for $[\![\_]\!]^{a\sharp}$ using constructive Galois connections. The state-of-the art approach of "γ-only" verification would instead mechanize the $\gamma\gamma$ variant of correctness:

$$[\![ae]\!]^a_\wp(\gamma^z(i^\sharp_1, i^\sharp_2)) = \gamma^z([\![ae]\!]^{a\sharp}(i^\sharp_1, i^\sharp_2))$$

which is similar to our $\mu\mu$ rule:

$$\langle i_1, i_2 \rangle \in \mu^{z \times z}(i^\sharp_1, i^\sharp_2) \wedge \langle i'_1, i'_2 \rangle = [\![ae]\!]^a(i_1, i_2) \Leftrightarrow \langle i'_1, i'_2 \rangle \in \mu^{z \times z}([\![ae]\!]^{a\sharp}(i^\sharp_1, i^\sharp_2))$$

The benefit of our approach is that soundness and completeness properties which also mention extraction (η) can also be mechanized, like calculating abstract interpreters from their specification.

Abstracting Relations The verification of an abstract interpreter for relations is similar to the design for functions: induce a specification using the constructive Galois connection, and prove correctness w.r.t. the induced spec. The relations we abstract are $\Downarrow^a$, $\Downarrow^b$ and $\mapsto^c$, and we call their abstract interpreters $\mathcal{A}^\sharp$, $\mathcal{B}^\sharp$ and $\mathcal{C}^\sharp$. Rather than postulate the definitions of the abstract interpreters, we calculate them from their specifications, the results of which are sound and computable by construction. The arithmetic and boolean abstract interpreters are functions from abstract environments to abstract integers, and the abstract interpreter for commands computes the next abstract transition states of execution. (We only present select calculations for $\mathcal{A}^\sharp$; see our accompanying Agda development for each calculation in mechanized form, and sections 4.8, 4.9 and 4.10 for detailed calculations of binary arithmetic operators and conditional command expressions.) $\mathcal{A}^\sharp$ has type:

$$\mathcal{A}^\sharp[\_] \;:\; \mathtt{aexp} \to \mathtt{env}^\sharp \nearrow \mathbb{Z}^\sharp$$

To induce a spec for $\mathcal{A}^\sharp$, we first revisit the concrete semantics relation as a powerset-valued function, which we call $\mathcal{A}$:

$$\mathcal{A}[\_] \;:\; \mathtt{aexp} \to \mathtt{env} \to \wp(\mathbb{Z}) \qquad\qquad \mathcal{A}[ae](\rho) \;\coloneqq\; \{i \mid \rho \vdash ae \Downarrow^a i\}$$

The induced spec for $\mathcal{A}^\sharp$ is generated with the monadic bind operator, which we notate using Moggi's star notation $\_^*$:

$$pure(\eta^z)^*(\mathcal{A}[ae]^*(\mu^r(\rho^\sharp))) \subseteq pure(\mathcal{A}^\sharp[ae])(\rho^\sharp)$$

which unfolds to:

$$\{\eta^z(i) \mid \rho \in \mu^r(\rho^\sharp) \wedge \rho \vdash ae \Downarrow^a i\} \subseteq \{\mathcal{A}^\sharp[ae](\rho^\sharp)\}$$

To calculate $\mathcal{A}^\sharp$ we reason equationally from the spec on the left towards the singleton set on the right, and declare the result the definition of $\mathcal{A}^\sharp$. We do this by case analysis on ae; we show the cases for $ae = \texttt{rand}$ and $ae = x$ in Figure 4.3. Each calculation can also be written in monadic form, which is the style we mechanize; we repeat the variable case in monadic form in the figure.

Mechanized Calculation Our Agda calculation of $\mathcal{A}^\sharp$ strongly resembles the on-paper monadic one. We show the Agda proof code for abstract variable references in Figure 4.4. The first line is the top level definition site for the derivation of $\mathcal{A}^\sharp$ for the `Var` case. The `proof-mode` term is part of our "proof-mode" library which gives support for calculational reasoning in the form of Agda proof combinators with mixfix syntax. Statements surrounded by double square brackets $[[e]]$ restate the current proof state, which Agda will check is correct. Reasoning steps are employed through $\wr$ e $\int$ terms, which transform the proof state from the previous form to the next. The term [`focus-right` $[\cdot]$ *of* e] focuses the goal to the right of the outermost application, scoped between `begin` and `end`.

Using constructive Galois connections, our mechanized calculation closely follows Cousot's classical one, uses both η and μ mappings, and results in a verified, executable static analyzer. Such a result is not possible using classical Galois connections, due to the inability to encode α functions constructively.

Case $ae = \texttt{rand}$**:**

$\{\eta^z(i) \mid \rho \in \mu^r(\rho^\sharp) \wedge \rho \vdash \texttt{rand} \Downarrow^a i\}$

$= \{\eta^z(i) \mid \rho \in \mu^r(\rho^\sharp) \wedge i \in \mathbb{Z}\}$ ⟅ defn. of $\rho \vdash \texttt{rand} \Downarrow^a i$ ⟆

$\subseteq \{\eta^z(i) \mid i \in \mathbb{Z}\}$ ⟅ $\varnothing$ when $\mu^r(\rho^\sharp) = \varnothing$ ⟆

$\subseteq \{\texttt{any}\}$ ⟅ $\{\texttt{any}\}$ mon. w.r.t. $\sqsubseteq^z$ ⟆

$\triangleq \{\mathcal{A}^\sharp[\texttt{rand}](\rho^\sharp)\}$ ⟅ defining $\mathcal{A}^\sharp[\texttt{rand}]$ ⟆

Case $ae = x$**:**

$\{\eta^z(i) \mid \rho \in \mu^r(\rho^\sharp) \wedge \rho \vdash x \Downarrow^a i\}$

$= \{\eta^z(\rho(x)) \mid \rho \in \mu^r(\rho^\sharp)\}$ ⟅ defn. of $\rho \vdash x \Downarrow^a i$ ⟆

$= \{\eta^z(i) \mid i \in \mu^z(\rho^\sharp(x))\}$ ⟅ defn. of $\mu^r(\rho^\sharp)$ ⟆

$\subseteq \{\rho^\sharp(x)\}$ ⟅ Eq. CGC-Red ⟆

$\triangleq \{\mathcal{A}^\sharp[x](\rho^\sharp)\}$ ⟅ defining $\mathcal{A}^\sharp[x]$ ⟆

Case $ae = x$ **(Monadic):**

$pure(\eta^z)^*(\mathcal{A}[x]^*(\mu^r(\rho^\sharp)))$

$= pure(\lambda\rho.\eta^z(\rho(x)))^*(\mu^r(\rho^\sharp))$ ⟅ defn. of $\mathcal{A}[x]$ ⟆

$= pure(\eta^z)^*(\mu^{z*}(\rho^\sharp(x)))$ ⟅ defn. of $\mu^r(\rho^\sharp)$ ⟆

$\subseteq ret(\rho^\sharp(x))$ ⟅ Eq. CGC-Red ⟆

$\triangleq pure(\mathcal{A}^\sharp[x])(\rho^\sharp)$ ⟅ defining $\mathcal{A}^\sharp[x]$ ⟆

Figure 4.3: Case Study 1: Select Constructive Galois Connection Calculations

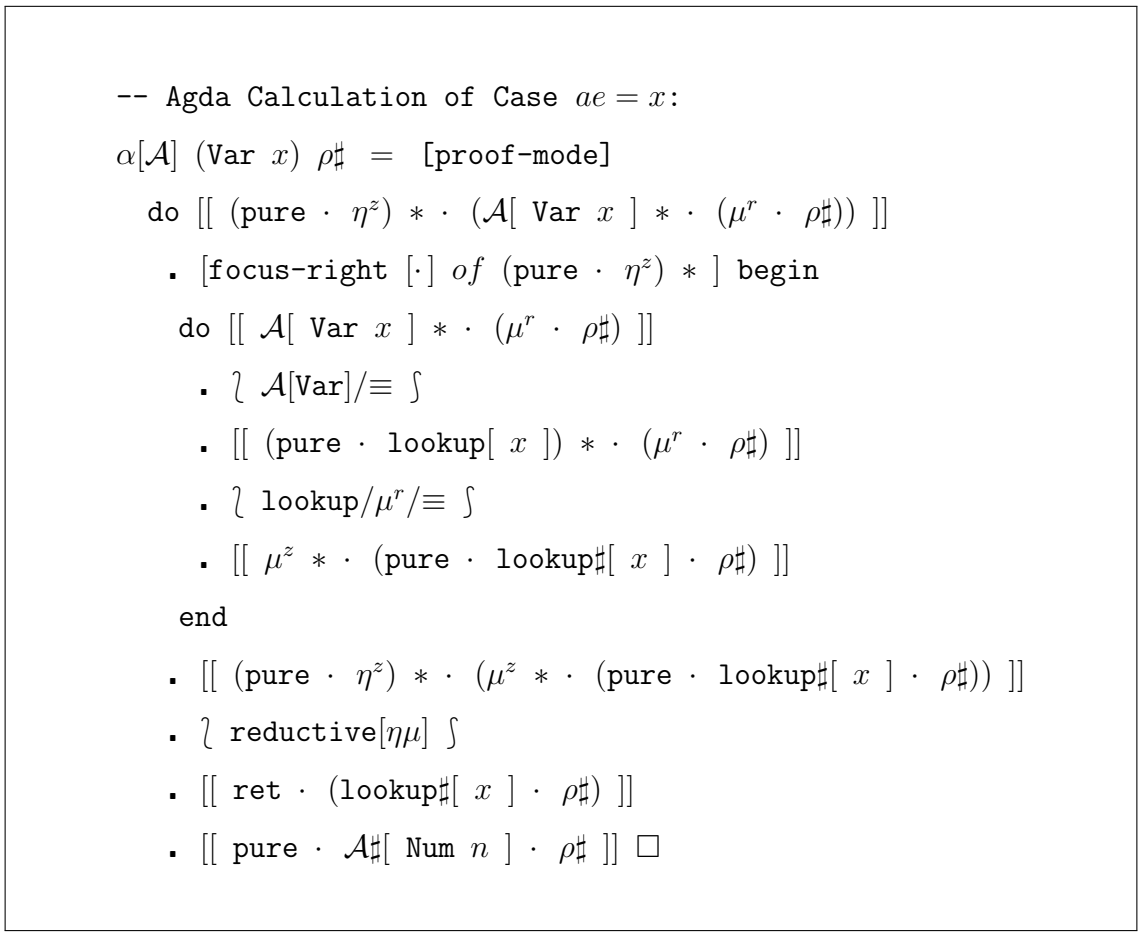


Figure 4.4: Case Study 1: Constructive Galois Connection Calculations in Agda

We complete the full calculation of Cousot's generic abstract interpreter for `WHILE` in Agda as supplemental material to this chapter, where the resulting interpreter is both sound and computable by construction. We also provide our "proof-mode" library which supports general calculational reasoning with posets.

The Classical Design Classically, one first designs a powerset lifting of the concrete semantics, called a *collecting semantics*:

$$\mathcal{A}_{\wp}[\_] \,:\, \texttt{aexp} \to \wp(\texttt{env}) \nearrow \wp(\mathbb{Z}) \qquad \mathcal{A}_{\wp}[ae](R) \coloneqq \{i \mid \rho \in R \wedge \rho \vdash ae \Downarrow^{a}\}$$

The classical soundness specification for $\mathcal{A}^{\sharp}[ae](\rho^{\sharp})$ is then:

$$\alpha^{z}(\mathcal{A}_{\wp}[ae](\gamma^{r}(\rho^{\sharp}))) \sqsubseteq \mathcal{A}^{\sharp}[ae](\rho^{\sharp})$$

However, as usual, the abstraction α^{z} cannot be mechanized effectively, preventing a mechanized derivation of $\mathcal{A}^{\sharp}$ by calculus.

4.5 Case Study 2: Gradual Type Systems

Recent work in metatheory for gradual type systems [Garcia et al., 2016] shows how a Galois connection discipline can guide the design of gradual typing systems. Starting with a Galois connection between precise and gradual types, both the static and dynamic semantics of the gradual language are derived systematically. This technique is called Abstracting Gradual Typing (AGT).

The design presented by Garcia *et al* is to begin with a precise type system, like the simply typed lambda calculus, and add a new type (?) which functions as

the top element ($\top$) in the lattice of type precision. The precise typing rules are presented with meta-operators for subtyping ($<:$) and for the join operator in the subtyping lattice ($\ddot{\vee}$). The gradual type system is then written using abstract variants of subtyping and join ($<:^\sharp$ and $\ddot{\vee}^\sharp$) which are proven correct w.r.t. specifications induced by the Galois connection.

The Precise Type System The AGT paper describes two designs for gradual type systems in increasing complexity. We chose to mechanize a hybrid of the two which is simple, like the first design, yet still exercises key challenges addressed by the second. We also made slight modifications to the design at parts to make mechanization easier, but without changing the nature of the system.

The precise type system we mechanized is the simply typed lambda calculus with booleans, and top and bottom elements for a subtyping lattice, which we call `any` and `none`:

$$\tau \in \texttt{type} ::= \texttt{none} \mid \mathbb{B} \mid \tau \to \tau \mid \texttt{any}$$

The first design in the AGT paper does not involve subtyping, and their second design incorporates record types with width and depth subtyping. By just focusing on `none` and `any`, we exercise the subtyping machinery of their approach without the blowup in complexity from formalizing record types.

The typing rules in AGT are written in strictly syntax-directed form, with explicit use of subtyping in rule hypotheses. We show three precise typing rules for if-statements, application and coercion in Figure 4.5. The subtyping lattice in the

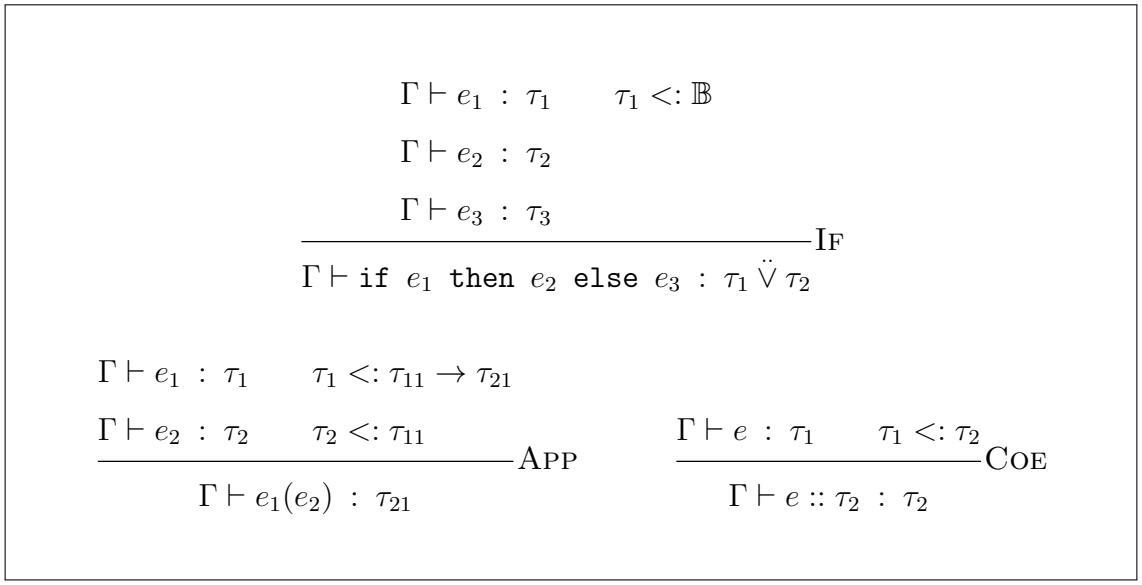


Figure 4.5: Case Study 2: Syntax Directed Precise Type System

precise system is the "safe for substitution" lattice, and well typed programs enjoy progress and preservation.

GRADUAL TYPES The essence of AGT is to design a gradual type system by *abstract interpretation* of the precise type system. To do this, a new top element is added to the precise type system, although rather than representing the top of the *subtyping* lattice like `any`, it represents the top of the *precision* lattice, and is notated ?:

$$\tau^\sharp \in \texttt{type}^\sharp \ ::= \ \texttt{none} \mid \mathbb{B} \mid \tau^\sharp \to \tau^\sharp \mid \texttt{any} \mid ?$$

The partial ordering has ? at the top ($\tau^\sharp \sqsubseteq ?$) and is otherwise discrete, and arrow types are monotonic (covariant) in both the domain and codomain:

$$\tau_{11}^\sharp \sqsubseteq \tau_{12}^\sharp \wedge \tau_{21}^\sharp \sqsubseteq \tau_{22}^\sharp \implies \tau_{11}^\sharp \to \tau_{21}^\sharp \sqsubseteq \tau_{12}^\sharp \to \tau_{22}^\sharp$$

Just as in our other designs by abstract interpretation, $\texttt{type}^\sharp$ is given meaning by an interpretation function μ, which is the constructive analog of a classical concretization (γ) function:

$$\mu \ : \ \texttt{type}^\sharp \nearrow \wp(\texttt{type}) \qquad \begin{array}{rcl} \mu(\tau^\sharp) & \coloneqq & \tau \quad \textit{when} \quad \tau^\sharp = \tau \in \{\texttt{none}, \mathbb{B}, \texttt{any}\} \\ \mu(\tau_1^\sharp \to \tau_2^\sharp) & \coloneqq & \{\tau_1 \to \tau_2 \mid \tau_1 \in \mu(\tau_1^\sharp) \wedge \tau_2 \in \mu(\tau_2^\sharp)\} \\ \mu(?) & \coloneqq & \{\tau \mid \tau \in \mathit{type}\} \end{array}$$

The extraction function η is, remarkably, the identity function:

$$\eta \ : \ \texttt{type} \to \texttt{type}^\sharp \qquad\qquad \eta(\tau) = \tau$$

and the constructive Galois correspondence holds:

$$\tau \in \mu(\tau^\sharp) \iff \eta(\tau) \sqsubseteq \tau^\sharp$$

Gradual Operators Given the constructive Galois connection between gradual and precise types, we synthesize specifications for abstract analogs of subtyping ($<:$) and the subtyping join operator ($\ddot{\vee}$), and relate them to their abstractions ($<:^\sharp$ and $\ddot{\vee}^\sharp$):

$$\tau_1 \in \mu(\tau_1^\sharp) \wedge \tau_2 \in \mu(\tau_2^\sharp) \wedge \tau_1 <: \tau_2 \iff \tau_1^\sharp <:^\sharp \tau_2^\sharp$$

$$\tau_1 \in \mu(\tau_1^\sharp) \wedge \tau_2 \in \mu(\tau_2^\sharp) \wedge \tau_3^\sharp \sqsubseteq \eta(\tau_1 \ddot{\vee} \tau_2) \iff \tau_3^\sharp \sqsubseteq \tau_1^\sharp \ddot{\vee}^\sharp \tau_2^\sharp$$

Key properties of gradual subtyping and the gradual join operator is how they operate over the unknown type ?:

$$? <:^\sharp \tau^\sharp \qquad\qquad \tau^\sharp <:^\sharp ? \qquad\qquad ? \ddot{\vee}^\sharp \tau^\sharp = \tau^\sharp \ddot{\vee}^\sharp ? = ?$$

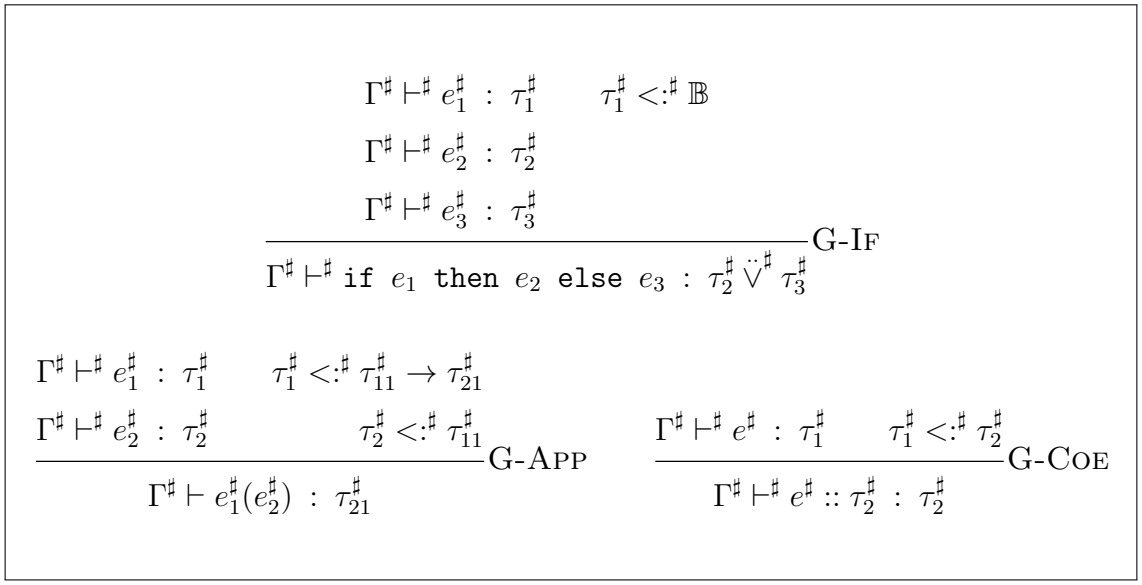


Figure 4.6: Case Study 2: Systematically Constructed Gradual Type System

Gradual Metatheory Using AGT, the gradual type system is a syntactic analog to the precise one but with gradual types and operators, which we show in Figure 4.6. Using this system, and constructive Galois connections, we mechanize in Agda the key AGT metatheory results from the paper: equivalence for fully-annotated terms (FAT), embedding of dynamic language terms (EDL), and the gradual guarantee (GG):

$$\vdash e : \tau \iff \vdash^\sharp e : \tau \tag{FAT}$$

$$closed(un) \implies \vdash^\sharp \lceil un \rceil : ? \tag{EDL}$$

$$\vdash^\sharp e_1^\sharp : \tau_1^\sharp \wedge e_1^\sharp \sqsubseteq e_2^\sharp \implies \vdash^\sharp e_2^\sharp : \tau_2^\sharp \wedge \tau_1^\sharp \sqsubseteq \tau_2^\sharp \tag{GG}$$

Adjunction	classical GCs	Kleisli GCs
Category	posets	posets
Adjoints	monotonic functions	monotonic $\wp$-monadic functions
Left Adjoint	$\alpha \ : \ A \nearrow B$	$\kappa\alpha \ : \ A \nearrow \wp(B)$
Right Adjoint	$\gamma \ : \ B \nearrow A$	$\kappa\gamma \ : \ B \nearrow \wp(A)$
Correspondence	$id(x) \sqsubseteq \gamma(y)$ $\iff$ $\alpha(x) \sqsubseteq id(y)$	$ret(x) \subseteq \kappa\gamma(y)$ $\iff$ $\kappa\alpha(x) \subseteq ret(y)$
Extensive	$id \sqsubseteq \gamma \circ \alpha$	$ret \sqsubseteq \kappa\gamma \circledast \kappa\alpha$
Reductive	$\alpha \circ \gamma \sqsubseteq id$	$\kappa\alpha \circledast \kappa\gamma \sqsubseteq ret$
Soundness	$\alpha \circ f \circ \gamma \sqsubseteq f^\sharp$	$\kappa\alpha \circledast f \circledast \kappa\gamma \sqsubseteq f^\sharp$
Optimality	$\alpha \circ f \circ \gamma = f^\sharp$	$\kappa\alpha \circledast f \circledast \kappa\gamma = f^\sharp$

Figure 4.7: Comparison of Constructive and Classical Galois Connection Adjunctions

4.6 Constructive Galois Connection Metatheory

In this section we develop the full metatheory of constructive Galois connection and prove precise claims about their relationship to classical Galois connections. We develop the metatheory of constructive Galois connections as an adjunction between posets with powerset-Kleisli adjoint functors. This is in contrast to classical Galois connections which come from an identical setup, but with the monotonic function space as adjoint functors, as shown in Figure 4.7.

We connect constructive to classical Galois connections through an isomorphism between a subset of classical to the entire space of constructive. To form this isomorphism we introduce an intermediate structure, Kleisli Galois connections,

which we show are isomorphic to the classical subset, and isomorphic to constructive ones. This second isomorphism uses the constructive *theorem* of choice, as depicted in Figure 4.8.

Classical Galois Connections We review classical Galois connections in Figure 4.7. A Galois connection between posets A and B contains two adjoint functors α and γ which share a correspondence. An equivalent formulation of the correspondence is two unit equations called extensive and reductive. Abstract interpreters are sound by over-approximating a specification induced by α and γ.

Powerset Monad See Sections 4.3.1 and 4.3.3 for the downward-closure monotonicity property, and monad definitions and notation for the monotonic powerset monad. The monad operators obey standard monad laws. We introduce one new operator for monadic function composition: $(g \circledast f)(x) \coloneqq g^*(f(x))$.

Kleisli Galois Connections We summarize Kleisli Galois connections in Figure 4.7. Kleisli Galois connections are analogous to classical ones, but with monadic analogs to α and γ, and monadic identity and composition operators ret and $\circledast$ in place of the function space identity and composition operators id and $\circ$.

Kleisli to Classical and Back All Kleisli Galois connections $\langle \kappa\alpha, \kappa\gamma \rangle$ between A and B can be lifted to recover a classical Galois connection $\langle \alpha, \gamma \rangle$ between $\wp(A)$ and $\wp(B)$ through a monadic lifting operator on Kleisli Galois connections $\langle \kappa\alpha, \kappa\gamma \rangle^*$:

$$\langle \alpha, \gamma \rangle \triangleq \langle \kappa\alpha, \kappa\gamma \rangle^* \coloneqq \langle \kappa\alpha^*, \kappa\gamma^* \rangle$$

This lifting is *sound*, meaning Kleisli soundness and optimality results can be translated to classical ones.

Theorem 1 (KGC-Sound$^{\texttt{AGDA}\checkmark}$)**.** *For any Kleisli relationship of soundness between f and $f^\sharp$, that is $\kappa\alpha \circledast f \circledast \kappa\gamma \sqsubseteq f^\sharp$, its lifting to classical is also sound, that is $\alpha \circ f^* \circ \gamma \sqsubseteq f^{\sharp *}$ where $\langle \alpha, \gamma \rangle = \langle \kappa\alpha, \kappa\gamma \rangle^*$, and likewise for optimality relationships $\alpha \circledast f \circledast \kappa y = f^\sharp$.*

This lifting is also *complete*, meaning classical Galois connection soundness and optimality results can always be translated to Kleisli ones, when α and γ are of lifted form.

Theorem 2 (KGC-Complete$^{\texttt{AGDA}\checkmark}$)**.** *For any classical relationship of soundness between f^* and $f^{\sharp *}$, that is $\alpha \circ f^* \circ \gamma \sqsubseteq f^{\sharp *}$, its lowering to Kleisli is also sound when $\langle \alpha, \gamma \rangle = \langle \kappa\alpha, \kappa\gamma \rangle^*$, that is $\kappa\alpha \circledast f \circledast \kappa\gamma \sqsubseteq f^\sharp$, and likewise for optimality relationships $\alpha \circ f^* \circ \gamma = f^{\sharp *}$.*

Due to soundness and completeness, one can work with the simpler setup of Kleisli Galois connections without any loss of generality. The setup is simpler because Kleisli Galois connection theorems only quantify over individual elements rather than elements of powersets. For example, the soundness criteria $\kappa\alpha \circledast f \circledast \kappa\gamma \sqsubseteq f^\sharp$ is proved by showing $\kappa\alpha^*(f^*(\kappa\gamma(x))) \subseteq f^\sharp(x)$ for an arbitrary element $x : A$, whereas in the classical proof one must show $\kappa\alpha^*(f^*(\kappa\gamma^*(X))) \subseteq f^{\sharp *}(X)$ for arbitrary sets $X : \wp(A)$.

Constructive Galois Connections Constructive Galois connections are a restriction of Kleisli Galois connections where the abstraction mapping is a pure rather than monadic function. We call the left adjoint *extraction*, notated η, and the right adjoint *interpretation*, notated μ. The constructive Galois connection correspondence, alternative expansive and reductive formulation of the correspondence, and soundness and optimality criteria are identical to Kleisli Galois connections where $\langle \kappa\alpha, \kappa\gamma \rangle = \langle pure(\eta), \mu \rangle$.

Constructive to Kleisli and Back Our main theorem which justifies the soundness and completeness of constructive Galois connections is an isomorphism between constructive and Kleisli Galois connections. The easy direction is soundness, where a Kleisli Galois connection is formed by defining $\langle \kappa\alpha, \kappa\gamma \rangle \coloneqq \langle pure(\eta), \mu \rangle$. Soundness and optimality theorems are then lifted from constructive to Kleisli without modification.

Theorem 3 (CGC-Sound$^{\texttt{AGDA}\checkmark}$)**.** *For any constructive relationship of soundness between f and $f^\sharp$, that is $pure(\eta) \circledast f \circledast \mu \sqsubseteq f^\sharp$, its lifting to Kleisli is sound, that is $\kappa\alpha \circledast f \circledast \kappa\gamma \sqsubseteq f^\sharp$ where $\langle \kappa\alpha, \kappa\gamma \rangle = \langle pure(\eta), \mu \rangle$, and likewise for optimality relationships $pure(\eta) \circledast f \circledast \mu = f^\sharp$.*

The other direction, completeness, is much more surprising. First we establish a lowering for Kleisli Galois connections.

Lemma 1 (CGC-Induce$^{\texttt{AGDA}\checkmark}$)**.** *For every Kleisli Galois connection $\langle \kappa\alpha, \kappa\gamma \rangle$, there exists a constructive Galois connection $\langle \eta, \mu \rangle$ where $\langle pure(\eta), \mu \rangle = \langle \kappa\alpha, \kappa\gamma \rangle$.*

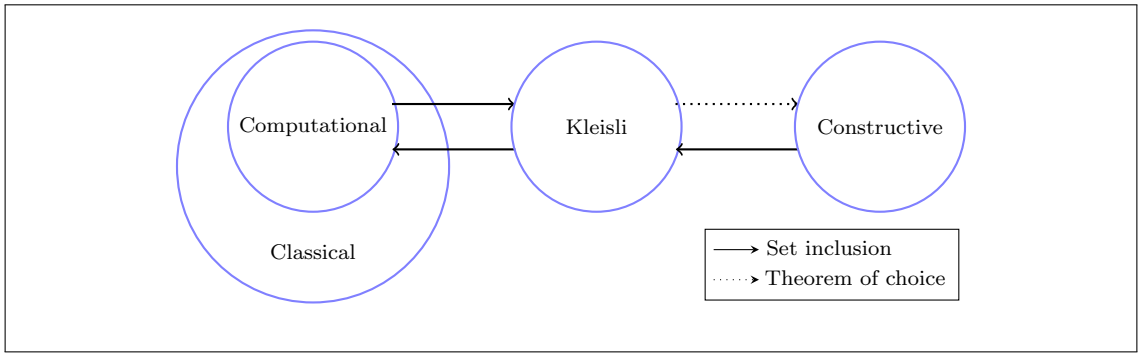


Figure 4.8: Relationship Between Classical, Kleisli and Constructive GCs

Proof. Because the mapping from Kleisli to constructive is interesting we provide a proof, which to our knowledge is novel. The proof builds a constructive Galois connection $\langle \eta, \mu \rangle$ from a Kleisli $\langle \kappa\alpha, \kappa\gamma \rangle$ by exploiting the Kleisli correspondence and making use of the constructive theorem of choice.

To turn an arbitrary Kleisli Galois connection into a constructive one, we show that the effect on $\kappa\alpha : A \nrightarrow \wp(B)$ is benign, or in other words, that there exists some η such that $\kappa\alpha = \mathit{pure}(\eta)$. We prove this using two ingredients: a constructive interpretation of the Kleisli extensive law, and the constructive *theorem* of choice.

We first expand the Kleisli expansive property, unfolding definitions of $\circledast$ and ret, to get an equivalent logical statement:

$$\forall x.\exists y.y \in \kappa\alpha(x) \wedge x \in \kappa\gamma(y) \qquad \text{(KGC-Exp)}$$

Statements of this form can be used in conjunction with an axiom of choice in classical mathematics, which is:

$$(\forall x.\exists y.R(x,y)) \implies \exists f.\forall x.R(x,f(x)) \qquad \text{(AxChoice)}$$

This theorem is admitted as an *axiom* in classical mathematics, but in constructive

logic—the setting used for extracting verified algorithms–(AxChoice) is definable as a *theorem*, due to the computational interpretation of logical connectives $\forall$ and $\exists$. We define (AxChoice) as a theorem in Agda without trouble:

$$\begin{aligned}
&\texttt{choice} : \forall\ \{A\ B\}\ \{R : A \to B \to \texttt{Set}\} \\
&\quad \to (\forall\ x \to \exists\ y\ st\ R\ x\ y) \\
&\quad \to (\exists\ f\ st \forall\ x \to R\ x\ (f\ x)) \\
&\texttt{choice}\ P = \langle \exists\ (\lambda\ x \to \pi_1\ (P\ x))\ ,\ (\lambda\ x \to \pi_2\ (P\ x))\ \rangle
\end{aligned}$$

Applying (AxChoice) to (KGC-Exp) then gives:

$$\exists \eta. \forall x. \eta(x) \in \kappa\alpha(x) \wedge x \in \kappa\gamma(\eta(x)) \tag{ExpChioce}$$

which proves the existence of a pure function $\eta : A \nrightarrow B$.

In order to form a constructive Galois connection η and μ must satisfy the correspondence, which we prove in split form:

$$x \in \mu(\eta(x)) \tag{CGC-Exp}$$

$$x \in \mu(y) \implies \eta(x) \sqsubseteq y \tag{CGC-Red}$$

The expansive property is immediate from the second conjunct in (ExpChioce). The reductive property follows from the Kleisli reductive property:

$$x \in \kappa\gamma(y) \wedge y' \in \kappa\alpha(x) \implies y' \sqsubseteq y \tag{KGC-Red}$$

The constructive variant of reductive is proved by satisfying the first two premises of (KGC-Red), where $x \in \kappa\gamma(y)$ is by assumption and $y' \in \kappa\alpha(x)$ is by the first conjunct in (ExpChioce).

So far we have shown that for a Kleisli Galois connection $\langle \kappa\alpha, \kappa\gamma \rangle$, there exists

a constructive Galois connection $\langle \eta, \mu \rangle$ where $\mu = \kappa\gamma$. However, we have yet to show $pure(\eta) = \kappa\alpha$. To show this, we prove an analog of a standard result for classical Galois connections: that α and γ uniquely determine each other.

Lemma 2 (Unique Abstraction$^{\texttt{AGDA}\checkmark}$)**.** *For any two Kleisli Galois connections* $\langle \kappa\alpha_1, \kappa\gamma_1 \rangle$ *and* $\langle \kappa\alpha_2, \kappa\gamma_2 \rangle$, $\kappa\alpha_1 = \kappa\alpha_2$ *iff* $\kappa\gamma_1 = \kappa\gamma_2$

We then conclude $pure(\eta) = \kappa\alpha$ as a consequence of the above lemma and the fact that $\mu = \kappa\gamma$.

□

Given the above mapping from Kleisli Galois connections to constructive ones, we prove the completeness of this mapping.

Theorem 4 (CGC-Complete$^{\texttt{AGDA}\checkmark}$)**.** *For any Kleisli relationship of soundness between* f *and* $f^\sharp$*, that is* $\kappa\alpha \circledast f \circledast \kappa\gamma \sqsubseteq f^\sharp$*, its lowering to constructive is also sound, that is* $pure(\eta) \circledast f \circledast \mu \sqsubseteq f^\sharp$ *where* $\langle \eta, \mu \rangle$ *is induced, and likewise for optimality relationships* $\kappa\alpha \circledast f \circledast \kappa\gamma = f^\sharp$.

Mechanization We mechanize the metatheory for constructive Galois connections and both case studies from Sections 4.4 and 4.5 in Agda, as well as a general purpose proof library for posets and calculational reasoning with the monotonic powerset monad. The development is available at: `github.com/plum-umd/cgc`.

Wrapping Up In this section we showed that constructive Galois connections are sound w.r.t. classical Galois connections, and complete w.r.t. the subset of

classical Galois connections recovered by lifting constructive ones. We showed this by introducing an intermediate space of Galois connections called Kleisli Galois connections, and by establishing two sets of isomorphisms between a subset of classical and Kleisli, and between Kleisli and constructive. The proof of isomorphism between constructive and Kleisli yielded an interesting proof which applies the constructive theorem of choice to one of the Kleisli Galois connection correspondence laws.

4.7 Constructing Constructive Galois Connections

The classical Galois connection framework comes with a library of connectives which are used to build larger Galois connections out of smaller, primitive ones [Cousot and Cousot, 1994]. For example, it is common to create a Galois connection for Cartesian products ($A \times B$) as the product abstraction of two Galois connections, one for each side (A and B).

In this section, we define the constructive analog of many classical Galois connection connectives and primitives. In later sections we will highlight similarities and differences between constructive and classical calculations (§ 4.8), how derivations of optimal abstract interpreters varies between the two settings (§ 4.9), and how multivalued computations are supported in the constructive setting (§ 4.10). Each section will make use of the connectives and primitives defined in this section without explicit introduction. Some readers may choose to skip this section, and refer back to the definitions as each connective appears in later sections.

By convention, we notate *classical* Galois connections $A \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} B$, that is with α and γ symbols below and above the arrows, and constructive Galois connections $A \underset{\eta}{\overset{\mu}{\leftrightarrows}} B$, that is with η and μ symbols below and above the arrows. Note that in the case of classical Galois connections, the domain and codomain of abstraction (α) and concretization (γ) are immediate from the notation, that is, $\alpha : A \nearrow B$ and $\gamma : B \nearrow A$. However for constructive Galois connections, the domain and codomain is only immediate from the notation for abstraction (η), but not concretization (μ) which maps to a powerset in the codomain, that is $\eta : A \nearrow B$ but $\mu : B \nearrow \wp(A)$. We notate $pure(x)$ compactly as $\lfloor x \rfloor$, and *assume all powersets are downward closed.*

4.7.1 Strictly Classical Galois Connections

Independent Attributes Abstraction The independent attributes abstraction is defined for relations ($\wp(A \times B)$), and constructs the classical Galois connection:

$$\wp(A \times B) \underset{\overset{IA}{\alpha}}{\overset{\overset{IA}{\gamma}}{\leftrightarrows}} \wp(A) \times \wp(B) \qquad \begin{array}{l} \overset{IA}{\alpha} : \wp(A \times B) \nearrow \wp(A) \times \wp(B) \\ \overset{IA}{\gamma} : \wp(A) \times \wp(B) \nearrow \wp(A \times B) \end{array}$$

$$\begin{array}{rcl} \overset{IA}{\alpha}(XY) & \coloneqq & \langle \{x \mid \exists y. \langle x, y \rangle \in XY\}, \{y \mid \exists x. \langle x, y \rangle \in XY\} \rangle \\ \overset{IA}{\gamma}(X, Y) & \coloneqq & \langle \{\langle x, y \rangle \mid x \in X \wedge y \in Y\} \rangle \end{array}$$

4.7.2 Strictly Constructive Galois Connections

Singleton Abstraction The singleton abstraction is defined for powersets of partially ordered sets ($\wp(A)$), and constructs the constructive Galois connection:

$$A \underset{\overset{1}{\eta}}{\overset{\overset{1}{\mu}}{\leftrightarrows}} \wp(A) \qquad \begin{array}{l} \overset{1}{\eta} : A \nearrow \wp(A) \\ \overset{1}{\mu} : \wp(A) \nearrow \wp(A) \end{array} \qquad \begin{array}{l} \overset{1}{\eta}(x) \coloneqq \{x\} \\ \overset{1}{\mu}(X) \coloneqq X \end{array}$$

4.7.3 Primitive Galois Connections—Classical and Constructive

Least-upper-bound Abstraction The least-upper-bound abstraction is defined for powersets of partially ordered sets ($\wp(A)$), and constructs the classical Galois connection:

$$\wp(A) \underset{\overset{\sqcup}{\alpha}}{\overset{\overset{\sqcup}{\gamma}}{\leftrightarrows}} A \qquad \begin{array}{ll} \overset{\sqcup}{\alpha} : \wp(A) \nearrow A & \overset{\sqcup}{\alpha}(X) \coloneqq \bigsqcup_{x \in X} x \\ \overset{\sqcup}{\gamma} : A \nearrow \wp(A) & \overset{\sqcup}{\gamma}(x) \coloneqq \{x\} \end{array}$$

The constructive analog is defined for powersets of partially ordered sets ($\wp(A)$), and constructs the *classical* Galois connection:

$$\wp(A) \underset{\overset{\sqcup_\wp}{\alpha}}{\overset{\overset{\sqcup_\wp}{\gamma}}{\leftrightarrows}} \wp^1(A) \qquad \begin{array}{ll} \overset{\sqcup_\wp}{\alpha} : \wp(A) \nearrow \wp^1(A) & \overset{\sqcup_\wp}{\alpha}(X) \coloneqq \{x \mid x \sqsubseteq \bigsqcup_{x \in X} x\} \\ \overset{\sqcup_\wp}{\gamma} : \wp^1(A) \nearrow \wp(A) & \overset{\sqcup_\wp}{\gamma}(X) \coloneqq \{x \mid x \in X\} \end{array}$$

We notate singleton (downward closed) powersets $\wp^1(\_)$, which classically are isomorphic to the carrier set ($\wp^1(A) \leftrightarrows A$), but not constructively.

Elementwise Abstraction The elementwise abstraction is defined given a function $f : A \to B$, and constructs the classical Galois connection:

$$\wp(A) \underset{\overset{[f]}{\alpha}}{\overset{\overset{[f]}{\gamma}}{\leftrightarrows}} \wp(B) \qquad \begin{array}{ll} \overset{[f]}{\alpha} : \wp(A) \nearrow \wp(B) & \overset{[f]}{\alpha}(X) \coloneqq \{f(x) \mid x \in X\} \\ \overset{[f]}{\gamma} : \wp(B) \nearrow \wp(A) & \overset{[f]}{\gamma}(Y) \coloneqq \{x \mid f(x) \in Y\} \end{array}$$

The constructive analog is defined given a *monotonic* function $f : A \nearrow B$ and constructs a constructive Galois connection $A \underset{\eta}{\overset{\mu}{\leftrightarrows}} B$ where:

$$\begin{array}{ll} \overset{[f]}{\eta} : A \nearrow B & \overset{[f]}{\eta}(x) \coloneqq f(x) \\ \overset{[f]}{\mu} : B \nearrow \wp(A) & \overset{[f]}{\mu}(y) \coloneqq \{x \mid f(x) \sqsubseteq y\} \end{array}$$

Fact 1 (Elementwise Abstraction Correspondence)**.** *The classical elementwise abstraction is equal to the classical lifting of the constructive elementwise abstraction,*

that is: $\alpha = \lfloor \eta \rfloor^*$ *and* $\gamma = \mu^*$.

4.7.4 Composing Galois Connections—Classical and Constructive

Abstraction Composition The composition of two abstractions is defined given abstractions $B \underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}} C$ and $A \underset{\alpha_2}{\overset{\gamma_2}{\leftrightarrows}} B$, and constructs the classical Galois connection:

$$A \underset{\overset{1\circ 2}{\alpha}}{\overset{\overset{1\circ 2}{\gamma}}{\leftrightarrows}} C \qquad \begin{array}{ll} \overset{1\circ 2}{\alpha} : A \nearrow C & \overset{1\circ 2}{\alpha}(x) \coloneqq \alpha_1(\alpha_2(x)) \\ \overset{1\circ 2}{\gamma} : C \nearrow A & \overset{1\circ 2}{\gamma}(z) \coloneqq \gamma_2(\gamma_1(z)) \end{array}$$

The constructive analog is defined given abstractions $B \underset{\eta_1}{\overset{\mu_1}{\leftarrow\!\!\!\rightarrow}} C$ and $A \underset{\eta_2}{\overset{\mu_2}{\leftarrow\!\!\!\rightarrow}} B$, and constructs the constructive Galois connection:

$$A \underset{\overset{1\circ 2}{\eta}}{\overset{\overset{1\circ 2}{\mu}}{\leftarrow\!\!\!\rightarrow}} C \qquad \begin{array}{ll} \overset{1\circ 2}{\eta} : A \nearrow C & \overset{1\circ 2}{\eta}(x) \coloneqq \eta_1(\eta_2(x)) \\ \overset{1\circ 2}{\mu} : C \nearrow \wp(A) & \overset{1\circ 2}{\mu}(z) \coloneqq \mu_2^*(\mu_1(z)) \end{array}$$

Product Abstraction The product abstraction is defined given abstractions $\wp(A) \underset{\alpha^A}{\overset{\gamma^A}{\leftrightarrows}} A^\sharp$ and $\wp(B) \underset{\alpha^B}{\overset{\gamma^B}{\leftrightarrows}} B^\sharp$, and constructs the classical Galois connection:

$$\wp(A) \times \wp(B) \underset{\overset{A\times B}{\alpha}}{\overset{\overset{A\times B}{\gamma}}{\leftrightarrows}} A^\sharp \times B^\sharp \qquad \begin{array}{ll} \overset{A\times B}{\alpha} : \wp(A) \times \wp(B) \nearrow A^\sharp \times B^\sharp \\ \overset{A\times B}{\gamma} : A^\sharp \times B^\sharp \nearrow \wp(A) \times \wp(B) \end{array}$$

$$\begin{array}{l} \overset{A\times B}{\alpha}(X, Y) \coloneqq \langle \alpha^A(X), \alpha^B(Y) \rangle \\ \overset{A\times B}{\gamma}(x^\sharp, y^\sharp) \coloneqq \langle \gamma^A(x^\sharp), \gamma^B(y^\sharp) \rangle \end{array}$$

The constructive analog is defined given abstractions $A \underset{\eta^A}{\overset{\mu^A}{\leftrightarrows}} A^\sharp$ and $B \underset{\eta^B}{\overset{\mu^B}{\leftrightarrows}} B^\sharp$, and constructs the constructive Galois connection:

$$A \times B \underset{\overset{A\times B}{\eta}}{\overset{\overset{A\times B}{\mu}}{\leftrightarrows}} A^\sharp \times B^\sharp \qquad \begin{array}{rcl} \overset{A\times B}{\eta} & : & A \times B \nearrow A^\sharp \times B^\sharp \\ \overset{A\times B}{\mu} & : & A^\sharp \times B^\sharp \nearrow \wp(A \times B) \end{array}$$

$$\begin{array}{rcl} \overset{A\times B}{\eta}(x, y) & \coloneqq & \langle \eta^A(x), \eta^B(y) \rangle \\ \overset{A\times B}{\mu}(x^\sharp, y^\sharp) & \coloneqq & \{\langle x, y \rangle \mid x \in \mu^A(x^\sharp) \wedge y \in \mu^B(y)\} \end{array}$$

Functional Abstraction The functional abstraction is defined given abstractions $\wp(A) \underset{\alpha^A}{\overset{\gamma^A}{\leftrightarrows}} A^\sharp$ and $\wp(B) \underset{\alpha^B}{\overset{\gamma^B}{\leftrightarrows}} B^\sharp$, and constructs the classical Galois connection:

$$\wp(A) \nearrow \wp(B) \underset{\overset{A\mapsto B}{\alpha}}{\overset{\overset{A\mapsto B}{\gamma}}{\leftrightarrows}} A^\sharp \nearrow B^\sharp \qquad \begin{array}{rcl} \overset{A\mapsto B}{\alpha} & : & (\wp(A) \nearrow \wp(B)) \nearrow A^\sharp \nearrow B^\sharp \\ \overset{A\mapsto B}{\gamma} & : & (A^\sharp \nearrow B^\sharp) \nearrow \wp(A) \nearrow \wp(A) \end{array}$$

$$\begin{array}{rcl} \overset{A\mapsto B}{\alpha}(f)(x^\sharp) & \coloneqq & \alpha^B(f(\gamma^A(x^\sharp))) \\ \overset{A\mapsto B}{\gamma}(f^\sharp)(X) & \coloneqq & \gamma^B(f^\sharp(\alpha^A(X))) \end{array}$$

The constructive analog is defined given constructive abstractions $A \underset{\eta^A}{\overset{\mu^A}{\leftrightarrows}} A^\sharp$ and $B \underset{\eta^B}{\overset{\mu^B}{\leftrightarrows}} B^\sharp$, and constructs the *classical* Galois connection:

$$A \nearrow \wp(B) \underset{\overset{A\overset{\wp}{\mapsto}B}{\alpha}}{\overset{\overset{A\overset{\wp}{\mapsto}B}{\gamma}}{\leftrightarrows}} A^\sharp \nearrow \wp(B^\sharp) \qquad \begin{array}{rcl} \overset{A\overset{\wp}{\mapsto}B}{\alpha} & : & (A \nearrow \wp(B)) \nearrow A^\sharp \nearrow \wp(B^\sharp) \\ \overset{A\overset{\wp}{\mapsto}B}{\gamma} & : & (A^\sharp \nearrow \wp(B^\sharp)) \nearrow A \nearrow \wp(B) \end{array}$$

$$\begin{array}{rcl} \overset{A\overset{\wp}{\mapsto}B}{\alpha}(f)(x^\sharp) & \coloneqq & \lfloor \eta^B \rfloor^*(f^*(\gamma^A(x^\sharp))) \\ \overset{A\overset{\wp}{\mapsto}B}{\gamma}(f^\sharp)(x) & \coloneqq & \mu^{B*}(f^\sharp(\eta^A(x))) \end{array}$$

Fact 2 (Functional Abstraction Correspondence)**.** *The classical functional abstraction is equal to the classical lifting of the constructive elementwise abstraction composed with the least-upper-bound abstraction, that is, for* $(f : A \nearrow \wp(B))$, $(f^\sharp : A^\sharp \nearrow B^\sharp)$,

$(X : \wp(A))$ *and* $(x^\sharp : A^\sharp)$*:*

$$\overset{A \mapsto B}{\alpha}(f^*)(x^\sharp) = \bigsqcup_{y^\sharp \in \overset{A \overset{\wp}{\to} B}{\alpha}(f)(x^\sharp)} y^\sharp \qquad \textit{and} \qquad \overset{A \mapsto B}{\gamma}(f^\sharp)(X) = \overset{A \overset{\wp}{\to} B}{\gamma}(\lfloor f^\sharp \rfloor)^*(X)$$

4.8 Comparing Classical and Constructive Approaches

In this section we aim to further clarify to what extent classical Galois connection calculations, which have been used successfully for decades, are related and/or inter-derivable with constructive Galois connection calculations. We will demonstrate this relationship between classical and constructive calculations through an extended example drawn from our first case study.

In Section 4.4 we showed calculations for the random number expression (`rand`) and variable reference (x). The inductive case for binary operators ($ae \oplus ae$) was omitted for brevity, however its calculation is particularly interesting because it involves interacting with a classical Galois connection during the calculation (in both constructive and classical settings). In this section we will work through this calculation in detail to demonstrate the differences and similarities between classical and constructive approaches, as well as to demonstrate the effectiveness of constructive Galois connections used in conjunction with classical ones.

SETUP To set the stage, we review in Figure 4.9 the types for the arithmetic operator denotation ($[\![\_]\!]^a$), its abstraction ($[\![\_]\!]^{a\sharp}$), the arithmetic expression relational semantics ($\_ \vdash \_ \Downarrow^a \_$), its functional variant ($\mathcal{A}[\_]$) and collecting semantics ($\mathcal{A}_\wp[\_]$), its abstraction ($\mathcal{A}^\sharp[\_]$), as well as classical and constructive Galois connections

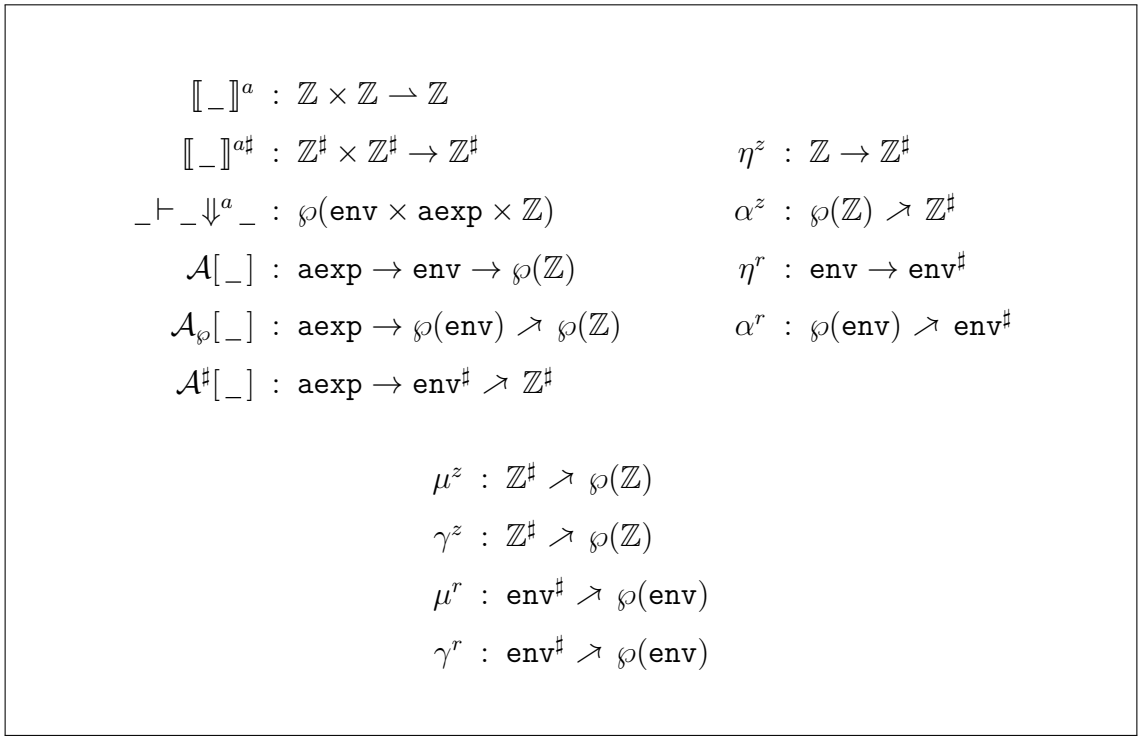


Figure 4.9: Review: Calculational Derivation for Binary Arithmetic Expressions

for integers ($\mathbb{Z} \underset{\eta^z}{\overset{\mu^z}{\leftrightarrows}} \mathbb{Z}^\sharp$ and $\mathbb{Z} \underset{\alpha^z}{\overset{\gamma^z}{\leftrightarrows}} \mathbb{Z}^\sharp$) and environments ($\mathtt{env} \underset{\eta^r}{\overset{\mu^r}{\leftrightarrows}} \mathtt{env}^\sharp$ and $\mathtt{env} \underset{\alpha^r}{\overset{\gamma^r}{\leftrightarrows}} \mathtt{env}^\sharp$).

First we will show the original classical calculation for binary arithmetic operator expressions which does not make explicit use of the independent attributes abstraction (§ 4.8.1). We will then make independent attributes explicit in the classical calculation (§ 4.8.2), and then show the constructive analog with explicit use of independent attributes (§ 4.8.3).

4.8.1 Review: Cousot's Original Classical Calculation

In the classical Galois connection framework, the abstraction ($\mathcal{A}^\sharp[\![\_]\!]$) for the arithmetic relational semantics ($\_ \vdash \_ \Downarrow^a \_$) is calculated by first defining the collecting

semantics ($\mathcal{A}_{\wp}[\_] : \texttt{aexp} \to \wp(\texttt{env}) \nearrow \wp(\mathbb{Z})$), and then relating the collecting semantics to the abstract semantics through a functional abstraction, that is:

$$^{r\mapsto z}\alpha(\mathcal{A}_{\wp}[ae])(\rho^\sharp) \triangleq \alpha^z(\mathcal{A}_{\wp}[ae](\gamma^r(\rho^\sharp))) \sqsubseteq \ldots \triangleq \mathcal{A}^\sharp[ae](\rho^\sharp)$$

Cousot's original calculation proceeds by case analysis on the syntax for arithmetic expressions, so for arithmetic operator expressions, the calculation is:

$$\alpha^z(\mathcal{A}_{\wp}[ae_1 \oplus ae_2](\gamma^r(\rho^\sharp))) \sqsubseteq \ldots \triangleq \mathcal{A}^\sharp[ae_1 \oplus ae_2](\rho^\sharp)$$

The calculation is shown in Figure 4.10. Steps 1–3 unfold semantic function and relation definitions; at Step 4 the specification is weakened explicitly to break the equality relationship between the environment used to evaluate ae_1 and ae_2; Step 5 rewrites the goal in terms of collecting semantics operations; Step 6 applies the inductive hypothesis; Step 7 applies a correct abstract interpreter for binary operators (a parameter to the calculation); Step 8 collapses neighboring abstraction and concretization functions; and Step 9 declares the final state of the calculation to be the definition of the algorithm.

Although there was no mention of the independent attributes abstraction in this calculation, its effects are there implicitly. In particular, Step 4, which breaks the equality relationship between environments, is implicitly performing the function of the independent attributes abstraction: to break relationships between elements of concrete sets of pairs. Step 4 is also the only step in the derivation which loses precision (uses $\sqsubseteq$ instead of $=$) unnecessarily, whereas the other losses of precision are unavoidable (inductive hypothesis, abstraction for binary operators, and collapsing

$$
\begin{array}{ll}
 & \alpha^z(\mathcal{A}_{\wp}[ae_1 \oplus ae_2](\gamma^r(\rho^\sharp))) \\
(1) & = \ \wr \text{ defn. of } \mathcal{A}_{\wp}[ae_1 \oplus ae_2] \ \int \\
 & \alpha^z(\bigcup_{\rho \in \gamma^r(\rho^\sharp)} \mathcal{A}[ae_1 \oplus ae_2](\rho)) \\
(2) & = \ \wr \text{ defn. of } \mathcal{A}[ae_1 \oplus ae_2] \ \int \\
 & \alpha^z(\bigcup_{\rho \in \gamma^r(\rho^\sharp)} \{[\![\oplus]\!]^a(i_1, i_2) \mid \rho \vdash ae_1 \Downarrow^a i_1 \wedge \rho \vdash ae_2 \Downarrow^a i_2\}) \\
(3) & = \ \wr \text{ defn. of } \mathcal{A}[ae_1] \text{ and } \mathcal{A}[ae_2] \ \int \\
 & \alpha^z(\bigcup_{\rho \in \gamma^r(\rho^\sharp)} \{[\![\oplus]\!]^a(i_1, i_2) \mid i_1 \in \mathcal{A}[ae_1](\rho) \wedge i_2 \in \mathcal{A}[ae_2](\rho)\}) \\
(4) & \sqsubseteq \ \wr \text{ monotonicity of } \alpha^z \ \int \\
 & \alpha^z(\bigcup_{\rho_1 \in \gamma^r(\rho^\sharp)} \bigcup_{\rho_2 \in \gamma^r(\rho^\sharp)} \{[\![\oplus]\!]^a(i_1, i_2) \mid i_1 \in \mathcal{A}[ae_1](\rho_1) \wedge i_2 \in \mathcal{A}[ae_2](\rho_2)\}) \\
(5) & = \ \wr \text{ set equality } \int \\
 & \alpha^z(\{[\![\oplus]\!]^a(i_1, i_2) \mid i_1 \in \mathcal{A}_{\wp}[ae_1](\gamma^r(\rho^\sharp)) \wedge i_2 \in \mathcal{A}_{\wp}[ae_2](\gamma^r(\rho^\sharp))\}) \\
(6) & \sqsubseteq \ \wr \text{ inductive hypothesis } (\mathcal{A}_{\wp}[ae] \circ \gamma^r \sqsubseteq \gamma^z \circ \mathcal{A}^\sharp[ae]) \ \int \\
 & \alpha^z(\{[\![\oplus]\!]^a(i_1, i_2) \mid i_1 \in \gamma^z(\mathcal{A}^\sharp[ae_1](\rho^\sharp)) \wedge i_2 \in \gamma^z(\mathcal{A}^\sharp[ae_2](\rho^\sharp))\}) \\
(7) & \sqsubseteq \ \wr \ [\![\oplus]\!]^{a\sharp} \text{ correct } ([\![\oplus]\!]^a_{\wp} \circ {}^{z\times z}\gamma^z \sqsubseteq \gamma^z \circ [\![\oplus]\!]^{a\sharp}) \ \int \\
 & \alpha^z(\gamma^z([\![\oplus]\!]^{a\sharp}(\mathcal{A}^\sharp[ae_1](\rho^\sharp), \mathcal{A}^\sharp[ae_2](\rho^\sharp)))) \\
(8) & \sqsubseteq \ \wr \ \alpha^z \circ \gamma^z \text{ reductive } (\alpha^z \circ \gamma^z \sqsubseteq id) \ \int \\
 & [\![\oplus]\!]^{a\sharp}(\mathcal{A}^\sharp[ae_1](\rho^\sharp), \mathcal{A}^\sharp[ae_2](\rho^\sharp)) \\
(9) & \triangleq \ \wr \text{ by } \mathcal{A}^\sharp[ae_1 \oplus ae_2](\rho^\sharp) \ := \ [\![\oplus]\!]^{a\sharp}(\mathcal{A}^\sharp[ae_1](\rho^\sharp), \mathcal{A}^\sharp[ae_2](\rho^\sharp)) \ \int \\
 & \mathcal{A}^\sharp[ae_1 \oplus ae_2](\rho^\sharp) \quad \blacksquare
\end{array}
$$

Figure 4.10: Classical Calculation for Binary Arithmetic Expressions

abstraction and concretization function). In the next subsection, we will make explicit use of the independent attributes abstraction, rather than through the ad-hoc line of reasoning contained in Step 4.

4.8.2 Using Independent Attributes Explicitly

In this section we recreate the calculation for binary arithmetic operator expressions from last section, but in a way that makes explicit use of the independent attributes abstraction.

The calculation is shown in Figure 4.11. The beginning of the derivation is as before (steps 1–3); Step 4.1 rewrites the calculation into a form that mentions independent attributes concretization; Step 4.2 pulls the collecting semantics for binary operators out of the union operation; Step 5.1 introduces the explicit independent attributes abstraction; Step 5.2 collapses the union operation between independent attributes abstraction and concretization based on a key observation (see below); Step 5.3 unfolds the definition of independent attributes concretization; and the rest of the derivation is as before (steps 6–9).

The key observation in this derivation is the fact that the independent attributes abstraction is transparent w.r.t. element-wise relationships, that is pairing $(\overset{IA}{\gamma})$ and splitting $(\overset{IA}{\alpha})$ two functions over related elements ($f(x_1)$ and $g(x_2)$ for $x_1 = x_2 \in X$), is equivalent to pairing each functions applied to unrelated elements ($f^*(X)$ and $g^*(X)$):

$\ldots$ *initial calculation as before (steps 1–3)*

$$\alpha^z(\bigcup_{\rho\in\gamma^r(\rho^\sharp)}\{[\![\oplus]\!]^a(i_1,i_2)\mid i_1\in\mathcal{A}[ae_1](\rho)\wedge i_2\in\mathcal{A}[ae_2](\rho)\})$$

(4.1) $= \wr$ defn. of $\overset{IA}{\gamma}$ and $[\![\oplus]\!]^a_\wp$ $\int$

$$\alpha^z(\bigcup_{\rho\in\gamma^r(\rho^\sharp)}[\![\oplus]\!]^a_\wp(\overset{IA}{\gamma}(\mathcal{A}[ae_1](\rho),\mathcal{A}[ae_2](\rho))))$$

(4.2) $= \wr$ set equality $\int$

$$\alpha^z([\![\oplus]\!]^a_\wp(\bigcup_{\rho\in\gamma^r(\rho^\sharp)}\overset{IA}{\gamma}(\mathcal{A}[ae_1](\rho),\mathcal{A}[ae_2](\rho))))$$

(5.1) $\sqsubseteq \wr$ $\overset{IA}{\gamma}\circ\overset{IA}{\alpha}$ extensive $(id\sqsubseteq\overset{IA}{\gamma}\circ\overset{IA}{\alpha})$ $\int$

$$\alpha^z([\![\oplus]\!]^a_\wp(\overset{IA}{\gamma}(\overset{IA}{\alpha}(\bigcup_{\rho\in\gamma^r(\rho^\sharp)}\overset{IA}{\gamma}(\mathcal{A}[ae_1](\rho),\mathcal{A}[ae_2](\rho))))))$$

(5.2) $= \wr$ set equality (see (IA-Split) below) $\int$

$$\alpha^z([\![\oplus]\!]^a_\wp(\overset{IA}{\gamma}(\mathcal{A}_\wp[ae_1](\gamma^r(\rho^\sharp)),\mathcal{A}_\wp[ae_2](\gamma^r(\rho^\sharp)))))$$

(5.3) $\sqsubseteq \wr$ defn. of $\overset{IA}{\gamma}$ and $[\![\oplus]\!]^a_\wp$ $\int$

$$\alpha^z(\{[\![\oplus]\!]^a(i_1,i_2)\mid i_1\in\mathcal{A}_\wp[ae_1](\gamma^r(\rho^\sharp))\wedge i_2\in\mathcal{A}_\wp[ae_2](\gamma^r(\rho^\sharp))\})$$

$\ldots$ *final calculation as before (steps 6–9)*

Figure 4.11: Classical Calculation for Binary Arithmetic Expressions Using Independent Attributes

Fact 3 (Independent Attributes Split Equality)**.**

$$\overset{IA}{\alpha}(\bigcup_{x \in X} \overset{IA}{\gamma}(f(x), g(x))) = \langle f^*(X), g^*(X) \rangle \qquad \text{(IA-Split)}$$

This observation captures locally the fact that if relational information is eventually going to be explicitly removed, then nothing is lost by splitting the equality relationship between arguments to each function.

One of the benefits of the calculational approach to abstract interpretation is that any loss of precision w.r.t. the induced specification is made explicit. In this derivation, the only non-essential loss in precision came from an explicit introduction of the independent attributes abstraction, which in turn makes explicit the fact that the resulting analysis is non-relational. If a relational analyzer was desired, one could point exactly where in the calculation this information was lost *via* the independent attributes abstraction, and correct it locally.

4.8.3 Calculating with Constructive Galois Connections

In the constructive framework, the abstract interpretation of binary arithmetic operator expressions ($\mathcal{A}^\sharp[ae_1 \oplus ae_2]$) is derived in a similar way, and also has the option of explicitly using the classical independent attributes abstraction along the way. The constructive calculation proceeds from the induced specification:

$$\overset{r \overset{\wp}{\mapsto} z}{\alpha}(\mathcal{A}[ae])(\rho^\sharp) \triangleq \lfloor \eta^z \rfloor^*(\mathcal{A}[ae]^*(\mu^r(\rho^\sharp))) \sqsubseteq \ldots \triangleq \lfloor \mathcal{A}^\sharp[ae] \rfloor (\rho^\sharp)$$

Two notable difference in the constructive calculation setup are:

1. The codomain type for both sides is $\wp(\mathbb{Z}^\sharp)$, not $\mathbb{Z}^\sharp$. This powerset modality

makes explicit the transition from "specification" to "algorithm."

2. The specification on the left-hand-side is *stronger* than the classical one, because it does not collapse the set of abstract integers $I^\sharp : \wp(\mathbb{Z}^\sharp)$ into a single least-upper-bound abstract integer $i^\sharp = \bigsqcup_{i^{\sharp\prime} \in I^\sharp} i^{\sharp\prime}$.

The original classical equation is recovered (in a constructive setting) by composing with the constructive least-upper-bound-abstraction $(\overset{\sqcup\wp}{\alpha} : \wp(\mathbb{Z}^\sharp) \nearrow \wp^1(\mathbb{Z}^\sharp))$:

$$\overset{\sqcup\wp}{\alpha}(\lfloor \eta^z \rfloor^*(\mathcal{A}[ae]^*(\mu^r(\rho^\sharp)))) \sqsubseteq \ldots \triangleq \lfloor \mathcal{A}^\sharp[ae] \rfloor(\rho^\sharp)$$

However, we will continue our demonstration with the original induced equation, where the constructive least-upper-bound-abstraction is not present.

The constructive calculation for the binary expression case proceeds in a similar fashion to Cousot's classical derivation. To mimic the classical derivation, the independent attributes abstraction is introduced to weaken the specification to discard the equality relationship between evaluation environments used to evaluate ae_1 and ae_2.

The calculation is shown in Figure 4.12. Steps 1–4 unfold semantic function and relation definitions; Step 5 explicitly weakens the specification using independent attributes; Step 6 applies the key independent attributes observation; Step 7 applies the inductive hypothesis; Step 8 combines concretization for independent attributes and the abstraction for integers; Step 9 applies a correct abstract interpreter for binary arithmetic operators (a parameter to the calculation); Step 10 collapses neighboring abstraction and concretization functions; and Step 11 declares the final

state of the calculation to be the definition of the algorithm.

What this calculation shows is that constructive Galois connections are able to work in tandem with classical Galois connections, as this constructive calculation made use of the classical independent attributes abstraction.

4.9 Optimal Calculations—Constructive and Classical

All of the derivations shown in the previous section follow a γ-directed approach to calculation. In this style, the next step of the calculation pushes concretization (γ) through the concrete semantics, from right to left, until it meets abstraction (α) on the far left-hand-side, at which point they collapse. In this section we explore the alternative approach of going the other direction: push abstraction from left-to-right until it meets concretization.

In the classical Galois connection framework, both γ-directed and α-directed approaches are similar, and the choice to use one or the other is mostly cosmetic. However, in the constructive framework, abstraction (η) is of a different nature than concretization (μ): it is a pure function with algorithmic content, rather than a relation. This means abstraction is easier to push through the concrete semantics, and therefore η-directed derivations can be simpler than η-directed ones.

Because constructive and classical Galois connections are so tightly connected, we show how this insight of η-directed calculations can be translated back to the world of classical Galois connections. To do this, we first make an observation about two restrictions often placed on collecting semantics and classical Galois connections

$$
\begin{array}{ll}
 & \lfloor \eta^z \rfloor^*(\mathcal{A}[ae_1 \oplus ae_2]^*(\mu^r(\rho^\sharp))) \\
(1) & = \ \wr \text{ defn. of } \mathcal{A}[ae_1 \oplus ae_2] \int \\
 & \lfloor \eta^z \rfloor^*(\bigcup_{\rho \in \mu^r(\rho^\sharp)} \{[\![\oplus]\!]^a(i_1, i_2) \mid \rho \vdash ae_1 \Downarrow^a i_1 \wedge \rho \vdash ae_2 \Downarrow^a i_2\}) \\
(2) & = \ \wr \text{ defn. of } \mathcal{A}[ae_1] \text{ and } \mathcal{A}[ae_2] \int \\
 & \lfloor \eta^z \rfloor^*(\bigcup_{\rho \in \mu^r(\rho^\sharp)} \{[\![\oplus]\!]^a(i_1, i_2) \mid i_1 \in \mathcal{A}[ae_1](\rho) \wedge i_2 \in \mathcal{A}[ae_2](\rho)\}) \\
(3) & = \ \wr \text{ defn. of } \overset{IA}{\gamma} \int \\
 & \lfloor \eta^z \rfloor^*(\bigcup_{\rho \in \mu^r(\rho^\sharp)} \lfloor [\![\oplus]\!]^a \rfloor^*(\overset{IA}{\gamma}(\mathcal{A}[ae_1](\rho), \mathcal{A}[ae_2](\rho)))) \\
(4) & = \ \wr \text{ set equality } \int \\
 & \lfloor \eta^z \rfloor^*(\lfloor [\![\oplus]\!]^a \rfloor^*(\bigcup_{\rho \in \mu^r(\rho^\sharp)} \overset{IA}{\gamma}(\mathcal{A}[ae_1](\rho), \mathcal{A}[ae_2](\rho)))) \\
(5) & \sqsubseteq \ \wr \ \overset{IA}{\gamma} \circ \overset{IA}{\alpha} \text{ extensive } (id \sqsubseteq \overset{IA}{\gamma} \circ \overset{IA}{\alpha}) \int \\
 & \lfloor \eta^z \rfloor^*(\lfloor [\![\oplus]\!]^a \rfloor^*(\overset{IA}{\gamma}(\overset{IA}{\alpha}(\bigcup_{\rho \in \mu^r(\rho^\sharp)} \overset{IA}{\gamma}(\mathcal{A}[ae_1](\rho), \mathcal{A}[ae_2](\rho)))))) \\
(6) & = \ \wr \text{ set equality (see (IA-Split) above) } \int \\
 & \lfloor \eta^z \rfloor^*(\lfloor [\![\oplus]\!]^a \rfloor^*(\overset{IA}{\gamma}(\mathcal{A}[ae_1]^*(\mu^r(\rho^\sharp)), \mathcal{A}[ae_2]^*(\mu^r(\rho^\sharp))))) \\
(7) & \sqsubseteq \ \wr \text{ inductive hypothesis } (\mathcal{A}[ae] \circledast \mu^r \sqsubseteq \mu^z \circledast \lfloor \mathcal{A}^\sharp[ae] \rfloor) \int \\
 & \lfloor \eta^z \rfloor^*(\lfloor [\![\oplus]\!]^a \rfloor^*(\overset{IA}{\gamma}(\mu^z(\mathcal{A}^\sharp[ae_1](\rho^\sharp)), \mu^z(\mathcal{A}^\sharp[ae_2](\rho^\sharp))))) \\
(8) & = \ \wr \text{ defn. of } \overset{IA}{\gamma} \text{ and } \overset{z \times z}{\mu} \int \\
 & \lfloor \eta^z \rfloor^*(\lfloor [\![\oplus]\!]^a \rfloor^*(\overset{z \times z}{\mu}(\mathcal{A}^\sharp[ae_1](\rho^\sharp), \mathcal{A}^\sharp[ae_2](\rho^\sharp)))) \\
(9) & \sqsubseteq \ \wr \ [\![\oplus]\!]^{a\sharp} \text{ correct } (\lfloor [\![\oplus]\!]^a \rfloor \circledast \overset{z \times z}{\mu} \sqsubseteq \mu^z \circledast \lfloor [\![\oplus]\!]^{a\sharp} \rfloor) \int \\
 & \lfloor \eta^z \rfloor^*(\mu^z([\![\oplus]\!]^{a\sharp}(\mathcal{A}^\sharp[ae_1](\rho^\sharp), \mathcal{A}^\sharp[ae_2](\rho^\sharp)))) \\
(10) & \sqsubseteq \ \wr \ \lfloor \eta^z \rfloor \circledast \mu^z \text{ reductive } (\lfloor \eta^z \rfloor \circledast \mu^z \sqsubseteq ret) \int \\
 & \{[\![\oplus]\!]^{a\sharp}(\mathcal{A}^\sharp[ae_1](\rho^\sharp), \mathcal{A}^\sharp[ae_2](\rho^\sharp))\} \\
(11) & \triangleq \ \wr \text{ by } \mathcal{A}^\sharp[ae_1 \oplus ae_2](\rho^\sharp) := [\![\oplus]\!]^{a\sharp}(\mathcal{A}^\sharp[ae_1](\rho^\sharp), \mathcal{A}^\sharp[ae_2](\rho^\sharp)) \int \\
 & \lfloor \mathcal{A}^\sharp[ae_1 \oplus ae_2] \rfloor(\rho^\sharp) \quad \blacksquare
\end{array}
$$

Figure 4.12: Constructive Calculation for Binary Arithmetic Expressions

in practice:

1. Restricting a predicate transformer ($t \ : \ \wp(A) \to \wp(B)$) to be a *complete union morphisms*, that is:

$$f(\bigcup_{i \in I} X_i) = \bigcup_{i \in I} (f(X_i))$$

for some indexed family of sets $X\_ \ : \ I \to \wp(A)$; and/or

2. Restricting an abstraction function ($\alpha \ : \ \wp(A) \nearrow A^\sharp$) is required to be a *complete join morphism*, that is:

$$\alpha(\bigcup_{i \in I} X_i) = \bigcup_{i \in I} (\alpha(X_i))$$

for some indexed family of sets $X\_ \ : \ I \to A^\sharp$

The main insight of this section is that the first property is equivalent to the existence of a monadic semantics relation, or $f \ : \ A \to \wp(B)$, where:

$$t(X) = \bigcup_{x \in X} f(x) \qquad \textit{and} \qquad f(x) = t(\{x\})$$

and the second property is equivalent to the existence of a constructive Galois connection, or $\eta \ : \ A \nearrow A^\sharp$, where:

$$\alpha(X) = \bigsqcup_{x \in X} \eta(x) \qquad \textit{and} \qquad \eta(x) = \alpha(\{x\})$$

It follows that, in any setting where classical Galois connections are used where the collecting semantics $t \ : \ \wp(A) \nearrow \wp(B)$ is a complete union morphism, and the abstraction functions $\alpha^A \ : \ \wp(A) \nearrow A^\sharp$ and $\alpha^B \ : \ \wp(B) \nearrow B^\sharp$ are complete join morphisms, it suffices to work purely with constructive Galois connections without

any loss of generality.

As a consequence of this, our observation above about η-directed calculations being easier to "push through" the calculation for constructive Galois connections also holds for α-directed classical calculations when the collecting semantics and abstraction function are both complete join/union morphisms.

The η-directed calculation of an abstract interpreter for binary arithmetic operator expressions is shown in Figure 4.13. The beginning of the calculation is as before (steps 1–2); Step 3 pushes the abstraction function through the union operation; Step 4 applies a correct abstract interpretation for binary operators (a parameter to the calculation); Step 5 pushes the abstraction function through the set comprehension; Step 6 applies the inductive hypothesis; Step 7 applies the fact that the abstract denotation for binary operators is monotonic, and that powerset are downward closed; Step 8 pushes abstraction again through the set comprehension; Step 9 collapses the neighboring abstraction and concretization functions; and Step 10 declares the final state of the calculation to be the definition of the algorithm.

This abstraction-directed calculation is not only simpler due to how easily the abstraction function distributes through powerset operations, but it is also optimal. Unlike the classical calculation (and the constructive μ-directed calculation), no loss in precision is explicitly introduced, and no use of independent attributes is made, explicitly or implicitly. Next, we show how to port this optimal calculation back to the classical Galois connection framework.

$$
\begin{array}{ll}
 & \ldots \textit{initial calculation as before (steps 1–2)} \\
 & \lfloor \eta^z \rfloor^* (\bigcup_{\rho \in \mu^r(\rho^\sharp)} \{ [\![\oplus]\!]^a(i_1, i_2) \mid i_1 \in \mathcal{A}[ae_1](\rho) \wedge i_2 \in \mathcal{A}[ae_2](\rho) \}) \\
\textit{(3)} & = \ \wr \text{ set equality } \int \\
 & \bigcup_{\rho \in \mu^r(\rho^\sharp)} \{ \eta^z([\![\oplus]\!]^a(i_1, i_2)) \mid i_1 \in \mathcal{A}[ae_1](\rho) \wedge i_2 \in \mathcal{A}[ae_2](\rho) \} \\
\textit{(4)} & \sqsubseteq \ \wr \ [\![\oplus]\!]^{a\sharp} \text{ correct } (\eta^z \circ [\![\oplus]\!]^a \sqsubseteq [\![\oplus]\!]^{a\sharp} \circ {}^{z\times z}\eta) \ \int \\
 & \bigcup_{\rho \in \mu^r(\rho^\sharp)} \{ [\![\oplus]\!]^{a\sharp}(\eta^z(i_1), \eta^z(i_2)) \mid i_1 \in \mathcal{A}[ae_1](\rho) \wedge i_2 \in \mathcal{A}[ae_2](\rho) \} \\
\textit{(5)} & = \ \wr \text{ set equality } \int \\
 & \bigcup_{\rho \in \mu^r(\rho^\sharp)} \{ [\![\oplus]\!]^{a\sharp}(i_1^\sharp, i_2^\sharp) \mid i_1^\sharp \in \lfloor \eta^z \rfloor^*(\mathcal{A}[ae_1](\rho)) \wedge i_2^\sharp \in \lfloor \eta^z \rfloor^*(\mathcal{A}[ae_2](\rho)) \} \\
\textit{(6)} & \sqsubseteq \ \wr \text{ inductive hypothesis } (\lfloor \eta^z \rfloor \circledast \mathcal{A}[ae] \sqsubseteq \lfloor \mathcal{A}^\sharp[ae] \rfloor \circledast \lfloor \eta^r \rfloor) \ \int \\
 & \bigcup_{\rho \in \mu^r(\rho^\sharp)} \{ [\![\oplus]\!]^{a\sharp}(i_1^\sharp, i_2^\sharp) \mid i_1^\sharp \sqsubseteq \mathcal{A}^\sharp[ae_1](\eta^r(\rho)) \wedge i_2^\sharp \sqsubseteq \mathcal{A}^\sharp[ae_1](\eta^r(\rho)) \} \\
\textit{(7)} & = \ \wr \text{ powerset downward-closed } \int \\
 & \bigcup_{\rho \in \mu^r(\rho^\sharp)} \{ [\![\oplus]\!]^{a\sharp}(\mathcal{A}^\sharp[ae_1](\eta^r(\rho)), \mathcal{A}^\sharp[ae_2](\eta^r(\rho))) \} \\
\textit{(8)} & = \ \wr \text{ powerset equality } \int \\
 & \bigcup_{\rho^{\sharp\prime} \in \lfloor \eta^r \rfloor^*(\mu^r(\rho^\sharp))} \{ [\![\oplus]\!]^{a\sharp}(\mathcal{A}^\sharp[ae_1](\rho^{\sharp\prime}), \mathcal{A}^\sharp[ae_2](\rho^{\sharp\prime})) \} \\
\textit{(9)} & \sqsubseteq \ \wr \ \lfloor \eta^r \rfloor \circledast \mu^r \text{ reductive } (\lfloor \eta^r \rfloor \circledast \mu^r \sqsubseteq ret) \ \int \\
 & \{ [\![\oplus]\!]^{a\sharp}(\mathcal{A}^\sharp[ae_1](\rho^\sharp), \mathcal{A}^\sharp[ae_2](\rho^\sharp)) \} \\
\textit{(10)} & \triangleq \ \wr \text{ by } \mathcal{A}^\sharp[ae_1 \oplus ae_2](\rho^\sharp) \ := \ [\![\oplus]\!]^{a\sharp}(\mathcal{A}^\sharp[ae_1](\rho^\sharp), \mathcal{A}^\sharp[ae_2](\rho^\sharp)) \ \int \\
 & \lfloor \mathcal{A}^\sharp[ae_1 \oplus ae_2] \rfloor (\rho^\sharp) \quad \blacksquare
\end{array}
$$

Figure 4.13: Constructive Calculation for Binary Arithmetic Expressions—Optimal and η-directed

Porting the Optimal Derivation Back to Classical In this η-directed constructive calculation, no steps lose precision unnecessarily. However, the classical calculation *required* an explicit loss of precision through the independent attributes abstraction. How can this be? To shed light on this question, we show that the constructive abstraction-directed calculation can be back-ported to a classical calculation, leveraging the fact that the abstraction side of Galois connections are complete join morphisms, that is:

$$\alpha^z(\bigcup_{i \in I} X_i) = \bigsqcup_{i \in I} (\alpha^z(X_i))$$

With this observation, a classical derivation is possible which doesn't need to interact with independent attributes to induce a final algorithm.

The classical calculation of binary arithmetic operator expressions is shown in Figure 4.14. The beginning of the calculation is as before (steps 1–3); Step 4 pushes abstraction through the union operation, due to being a complete join morphism; Step 5 applies a correct abstraction for binary operators; Step 6 applies the inductive hypothesis; Step 7 pulls abstraction out of the set comprehension; Step 8 pushes abstraction through the set comprehension, due to being a complete join morphism; Step 9 collapses adjacent abstraction and concretization functions; and Step 10 declares the final state of the calculation to be the definition of the algorithm.

$\ldots$ *initial calculation as before (steps 1–3)*

$$\alpha^z(\bigcup_{\rho\in\gamma^r(\rho^\sharp)}\{[\![\oplus]\!]^a(i_1,i_2)\mid i_1\in\mathcal{A}[ae_1](\rho)\wedge i_2\in\mathcal{A}[ae_2](\rho)\})$$

(4) $\quad\wr$ α^z complete join morphism $\int$

$$\bigsqcup_{\rho\in\gamma^r(\rho^\sharp)}\alpha^z(\{[\![\oplus]\!]^a(i_1,i_2)\mid i_1\in\mathcal{A}[ae_1](\rho)\wedge i_2\in\mathcal{A}[ae_2](\rho)\})$$

(5) $\quad\sqsubseteq\ \wr$ $[\![\oplus]\!]^{a\sharp}$ correct $(\alpha^z\circ[\![\oplus]\!]^a_\wp\overset{\sqsubseteq}{\sqsubseteq}[\![\oplus]\!]^{a\sharp}\circ{}^{z\times z}\alpha^z)$ $\int$

$$\bigsqcup_{\rho\in\gamma^r(\rho^\sharp)}[\![\oplus]\!]^{a\sharp}(\alpha^z(\mathcal{A}[ae_1](\rho)),\alpha^z(\mathcal{A}[ae_2](\rho)))$$

(6) $\quad\sqsubseteq\ \wr$ inductive hypothesis $(\alpha^z\circ\mathcal{A}_\wp[ae]=\mathcal{A}_\wp[ae]\circ\alpha^r)$ $\int$

$$\bigsqcup_{\rho\in\gamma^r(\rho^\sharp)}[\![\oplus]\!]^{a\sharp}(\mathcal{A}^\sharp[ae_1](\alpha^r(\{\rho\})),\mathcal{A}^\sharp[ae_2](\alpha^r(\{\rho\})))$$

(7) $\quad=\ \wr$ set equality $\int$

$$\bigsqcup_{\rho^{\sharp\prime}\in\{\alpha^r(\{\rho\})\mid\rho\in\gamma^r(\rho^\sharp)\}}[\![\oplus]\!]^{a\sharp}(\mathcal{A}^\sharp[ae_1](\rho^{\sharp\prime}),\mathcal{A}^\sharp[ae_2](\rho^{\sharp\prime}))$$

(8) $\quad=\ \wr$ α^r complete join morphism $\int$

$$\bigsqcup_{\rho^{\sharp\prime}\in\{\alpha^r(\gamma^r(\rho^\sharp))\}}[\![\oplus]\!]^{a\sharp}(\mathcal{A}^\sharp[ae_1](\rho^{\sharp\prime}),\mathcal{A}^\sharp[ae_2](\rho^{\sharp\prime}))$$

(9) $\quad\sqsubseteq\ \wr$ $\alpha^r\circ\gamma^r$ reductive $(\alpha^r\circ\gamma^r\sqsubseteq id)$ $\int$

$$[\![\oplus]\!]^{a\sharp}(\mathcal{A}^\sharp[ae_1](\rho^\sharp),\mathcal{A}^\sharp[ae_2](\rho^\sharp))$$

(10) $\quad\triangleq\ \wr$ by $\mathcal{A}^\sharp[ae_1\oplus ae_2](\rho^\sharp)\ :=\ [\![\oplus]\!]^{a\sharp}(\mathcal{A}^\sharp[ae_1](\rho^\sharp),\mathcal{A}^\sharp[ae_2](\rho^\sharp))$ $\int$

$$\mathcal{A}^\sharp[ae_1\oplus ae_2](\rho^\sharp)\quad\blacksquare$$

Figure 4.14: Classical Calculation for Binary Arithmetic Expressions—Optimal and α-directed

4.10 Multivalued Constructive Galois Connections

In this section we argue that constructive Galois connections support multivalued Galois connections, concrete semantics, and abstract interpreters, while maintaining their ability to be mechanized effectively.

To explore multivalued constructive Galois connections, we again work through an extended example based on the first case study, but this time deriving an abstract interpreter for conditional expressions (`if` *be* `then` *ce* `else` *ce*) in the command language (`cexp`) rather than arithmetic expressions (`aexp`).

SETUP To set the stage, we review in Figure 4.15 the types for the command expression relational semantics ($\_ \mapsto^c \_$), its functional variant ($\mathcal{C}[\_]$) and collecting semantics ($\mathcal{C}_\wp[\_]$), its abstraction ($\mathcal{C}^\sharp[\_]$), as well as classical and constructive Galois connections for integers ($\mathbb{Z} \underset{\eta^z}{\overset{\mu^z}{\leftrightarrows}} \mathbb{Z}^\sharp$ and $\mathbb{Z} \underset{\alpha^z}{\overset{\gamma^z}{\leftrightarrows}} \mathbb{Z}^\sharp$) and environments ($\texttt{env} \underset{\eta^r}{\overset{\mu^r}{\leftrightarrows}} \texttt{env}^\sharp$ and $\texttt{env} \underset{\alpha^r}{\overset{\gamma^r}{\leftrightarrows}} \texttt{env}^\sharp$).

4.10.1 Review: Cousot's Original Classical Calculation

In the classical Galois connection framework, the abstraction ($\mathcal{C}^\sharp[\_]$) for the command small-step relational semantics ($\_ \mapsto^c \_$) is calculated first by constructing the collecting semantics ($\mathcal{C}_\wp[\_]$), and then relating the collecting semantics to the abstract semantics through a functional abstraction, that is:

$$^{\Sigma \mapsto \Sigma}\alpha(\mathcal{C}_\wp[ce])(\Sigma^\sharp) \triangleq \alpha^\Sigma(\mathcal{C}_\wp[ce](\gamma^\Sigma(\Sigma^\sharp))) \sqsubseteq \ldots \triangleq \mathcal{C}^\sharp[ce](\Sigma^\sharp)$$

$$
\begin{array}{rcl@{\qquad\qquad}rcl}
\varsigma \in \Sigma & \coloneqq & \texttt{env} \times \texttt{cexp} & & & \\
\varsigma^\sharp \in \Sigma^\sharp & \coloneqq & \texttt{env}^\sharp \times \wp(\texttt{cexp}) & \eta^z & : & \mathbb{Z} \to \mathbb{Z}^\sharp \\
\_ \mapsto^c \_ & : & \wp(\Sigma \times \Sigma) & \alpha^z & : & \wp(\mathbb{Z}) \nearrow \mathbb{Z}^\sharp \\
\mathcal{C}[\![\_]\!] & : & \texttt{cexp} \to \Sigma \nearrow \wp(\Sigma) & \eta^r & : & \texttt{env} \to \texttt{env}^\sharp \\
\mathcal{C}_\wp[\![\_]\!] & : & \texttt{cexp} \to \wp(\Sigma) \nearrow \wp(\Sigma) & \alpha^r & : & \wp(\texttt{env}) \nearrow \texttt{env}^\sharp \\
\mathcal{C}^\sharp[\![\_]\!] & : & \texttt{cexp} \to \Sigma^\sharp \nearrow \Sigma^\sharp & & &
\end{array}
$$

$$
\begin{array}{rcl}
\mu^z & : & \mathbb{Z}^\sharp \nearrow \wp(\mathbb{Z}) \\
\gamma^z & : & \mathbb{Z}^\sharp \nearrow \wp(\mathbb{Z}) \\
\mu^r & : & \texttt{env}^\sharp \nearrow \wp(\texttt{env}) \\
\gamma^r & : & \texttt{env}^\sharp \nearrow \wp(\texttt{env})
\end{array}
$$

Figure 4.15: Review: calculating abstraction for conditional expressions

where configurations ($\varsigma \in \Sigma$) are abstracted through a composition of independent attributes and a product abstraction over environments:

$$
\wp(\Sigma) \underset{\overset{IA}{\alpha}}{\overset{\overset{IA}{\gamma}}{\leftrightarrows}} \wp(\texttt{env}) \times \wp(\texttt{cexp}) \underset{\overset{r \times id}{\alpha}}{\overset{\overset{r \times id}{\gamma}}{\leftrightarrows}} \Sigma^\sharp
\qquad
\begin{array}{rcl@{\qquad}rcl}
\alpha^\Sigma & : & \wp(\Sigma) \nearrow \Sigma^\sharp & \alpha^\Sigma & \coloneqq & \overset{r \times id}{\alpha} \circ \overset{IA}{\alpha} \\
\gamma^\Sigma & : & \Sigma^\sharp \nearrow \wp(\Sigma) & \gamma^\Sigma & \coloneqq & \overset{IA}{\gamma} \circ \overset{r \times id}{\gamma}
\end{array}
$$

In Cousot's original derivation, the abstract interpreter is derived for the reflexive transitive closure of the small step relation directly. We will instead present the abstract interpreter for the just the small step relation, factored out from the reflexive transitive closure.

The classical calculation begins by case analysis on the syntax for command expressions, so for conditional expressions the calculation is:

$$
\alpha^\Sigma(\mathcal{C}_\wp[\![\texttt{if } be \texttt{ then } ce_1 \texttt{ else } ce_2]\!](\gamma^r(\rho^\sharp))) \sqsubseteq \ldots \triangleq \mathcal{C}^\sharp[\![\texttt{if } be \texttt{ then } ce_1 \texttt{ else } ce_2]\!](\rho^\sharp)
$$

The calculation is shown in Figure 4.16. Steps 1–4 unfold semantic function and relation definitions; Step 5 weakens the specification through an (implicit) independent attributes abstraction; Step 6 applies a correct abstract interpreter for boolean expressions (a parameter to the calculation); Step 7 weakens the case when neither branch is valid, which would result in the returned abstract environment being bottom ($\bot$), or the empty map ($\varnothing$); Step 8 collapses adjacent abstraction and concretization functions; and Step 9 declares the final state of the calculation as the definition of the algorithm.

4.10.2 The Constructive Calculation

The goal is now to recreate this calculation using constructive Galois connections. Up until this point, the use of powersets has been entirely restricted to describing classical specifications. However, in this classical derivation, *finite* powersets appear in the resulting algorithm. Thus, powersets served double-duty: both for classical specification and for multivalued algorithmic results. When porting to constructive Galois connections, this distinction must be made explicit in order to support extraction of a verified algorithm.

Constructive Finite Sets To distinguish between classical powersets and algorithmic finite sets, we will continue to notate classical powersets as $\wp(A)$, which are modeled as downward-closed $A \searrow \mathit{prop}$. We will notate constructive finite sets as $\mathfrak{p}(A)$, which are representable in an algorithm using a data structure such as a sorted list, binary tree, or hashed dictionary. To distinguish classical powersets

$$
\begin{array}{ll}
 & \alpha^{\Sigma}(\mathcal{C}_{\wp}[\texttt{if}\ be\ \texttt{then}\ ce_1\ \texttt{else}\ ce_2](\gamma^r(\rho^\sharp))) \\
\textit{(1)} & = \ \wr \text{ defn. of } \mathcal{C}_{\wp}[\texttt{if}\ be\ \texttt{then}\ ce_1\ \texttt{else}\ ce_2] \int \\
 & \alpha^{\Sigma}(\bigcup_{\rho\in\gamma^r(\rho^\sharp)} \{\langle\rho', ce\rangle \mid \langle\rho, \texttt{if}\ be\ \texttt{then}\ ce_1\ \texttt{else}\ ce_2\rangle \mapsto^c \langle\rho', ce\rangle\}) \\
\textit{(2)} & = \ \wr \text{ defn. of } \langle\rho, \texttt{if}\ be\ \texttt{then}\ ce_1\ \texttt{else}\ ce_2\rangle \mapsto^c \langle\rho', ce'\rangle \int \\
 & \alpha^{\Sigma}(\bigcup_{\rho\in\gamma^r(\rho^\sharp)} \{\langle\rho, ce_1\rangle \mid \rho \vdash be \Downarrow^b true\} \cup \{\langle\rho, ce_2\rangle \mid \rho \vdash be \Downarrow^b false\}) \\
\textit{(3)} & = \ \wr \text{ defn. of } \rho \vdash be \Downarrow^b b \int \\
 & \alpha^{\Sigma}(\bigcup_{\rho\in\gamma^r(\rho^\sharp)} \{\langle\rho, ce_1\rangle \mid true = \mathcal{B}[be](\rho)\} \cup \{\langle\rho, ce_2\rangle \mid false = \mathcal{B}[be](\rho)\}) \\
\textit{(4)} & = \ \wr \text{ set equality (union commutativity) } \int \\
 & \alpha^{\Sigma}\left(\bigcup\left\{\begin{array}{l}\bigcup_{\rho\in\gamma^r(\rho^\sharp)} \{\langle\rho, ce_1\rangle \mid true = \mathcal{B}[be](\rho)\} \\ \bigcup_{\rho\in\gamma^r(\rho^\sharp)} \{\langle\rho, ce_2\rangle \mid false = \mathcal{B}[be](\rho)\}\end{array}\right.\right) \\
\textit{(5)} & \sqsubseteq \ \wr \text{ monotonicity (independent attributes) } \int \\
 & \alpha^{\Sigma}\left(\bigcup\left\{\begin{array}{l}\{\langle\rho, ce_1\rangle \mid \rho \in \gamma^r(\rho^\sharp) \wedge \exists\rho'.true = \mathcal{B}[be](\rho')\} \\ \{\langle\rho, ce_2\rangle \mid \rho \in \gamma^r(\rho^\sharp) \wedge \exists\rho'.false = \mathcal{B}[be](\rho')\}\end{array}\right.\right) \\
\textit{(6)} & \sqsubseteq \ \wr \ \mathcal{B}^\sharp[be] \text{ correct } (\mathcal{B}_{\wp}[be] \circ \gamma^r \sqsubseteq \gamma^b \circ \mathcal{B}^\sharp[be]) \int \\
 & \alpha^{\Sigma}\left(\bigcup\left\{\begin{array}{ll}\{\langle\rho, ce_1\rangle \mid \rho \in \gamma^r(\rho^\sharp)\} & if \quad true \sqsubseteq \mathcal{B}^\sharp[be](\rho^\sharp) \\ \{\langle\rho, ce_2\rangle \mid \rho \in \gamma^r(\rho^\sharp)\} & if \quad false \sqsubseteq \mathcal{B}^\sharp[be](\rho^\sharp)\end{array}\right.\right) \\
\textit{(7)} & \sqsubseteq \ \wr \text{ ignore case } \neg(true \sqsubseteq \mathcal{B}^\sharp[be](\rho^\sharp) \vee false \sqsubseteq \mathcal{B}^\sharp[be](\rho^\sharp)) \int \\
 & \left\langle \alpha^r(\gamma^r(\rho^\sharp)), \bigcup\left\{\begin{array}{ll}\{ce_1\} & if \quad true \sqsubseteq \mathcal{B}^\sharp[be](\rho^\sharp) \\ \{ce_2\} & if \quad false \sqsubseteq \mathcal{B}^\sharp[be](\rho^\sharp)\end{array}\right.\right\rangle \\
\textit{(8)} & \sqsubseteq \ \wr \ \alpha^r \circ \gamma^r \text{ reductive } (\alpha^r \circ \gamma^r \sqsubseteq id) \int \\
 & \left\langle \rho^\sharp, \bigcup\left\{\begin{array}{ll}\{ce_1\} & if \quad true \sqsubseteq \mathcal{B}^\sharp[be](\rho^\sharp) \\ \{ce_2\} & if \quad false \sqsubseteq \mathcal{B}^\sharp[be](\rho^\sharp)\end{array}\right.\right\rangle \\
\textit{(9)} & \triangleq \ \wr \text{ by } \mathcal{C}^\sharp[\texttt{if}\ be\ \texttt{then}\ ce_1\ \texttt{else}\ ce_2](\rho^\sharp) \coloneqq \left\langle \rho^\sharp, \bigcup\left\{\begin{array}{ll}\{ce_1\} & if \quad true \sqsubseteq \mathcal{B}^\sharp[be](\rho^\sharp) \\ \{ce_2\} & if \quad false \sqsubseteq \mathcal{B}^\sharp[be](\rho^\sharp)\end{array}\right.\right\rangle \int \\
 & \mathcal{C}^\sharp[\texttt{if}\ be\ \texttt{then}\ ce_1\ \texttt{else}\ ce_2](\rho^\sharp) \quad \blacksquare
\end{array}
$$

Figure 4.16: Classical Calculation for Conditional Command Expressions

from constructive finite sets notationally, we will continue to notate elements of powersets of posets $X : \wp(A)$ as $\{x \mid P(x)\}$, which is valid for any downward-closed proposition $P : A \searrow \mathit{prop}$, and notate elements of constructive finite sets $(\mathfrak{X} : \mathfrak{p}(A))$ as $\{\!\{x \mid P(x)\}\!\}$, which is valid for any *decidable* downward-closed proposition $P : A \searrow \mathbb{B}$.

We relate classical powersets $(\wp(A))$ to constructive finite sets $(\mathfrak{p}(A))$ using a constructive Galois connection:

$$\mathfrak{p}(A) \underset{\overset{\mathfrak{p}}{\eta}}{\overset{\overset{\mathfrak{p}}{\mu}}{\leftrightarrows}} \wp(A) \qquad \begin{array}{lll} \overset{\mathfrak{p}}{\eta} : \mathfrak{p}(A) \nearrow \wp(A) & & \overset{\mathfrak{p}}{\eta}(\mathfrak{X}) \coloneqq \{x \mid x \in X\} \\ \overset{\mathfrak{p}}{\mu} : \wp(A) \nearrow \wp(\mathfrak{p}(A)) & & \overset{\mathfrak{p}}{\mu}(X) \coloneqq \{\mathfrak{X} \mid \forall x.x \in X \Leftrightarrow x \in \mathfrak{X}\} \end{array}$$

and define a singleton abstraction for constructive finite sets:

$$A \underset{\overset{1\mathfrak{p}}{\eta}}{\overset{\overset{1\mathfrak{p}}{\mu}}{\leftrightarrows}} \mathfrak{p}(A) \qquad \begin{array}{lll} \overset{1\mathfrak{p}}{\eta} : A \nearrow \mathfrak{p}(A) & & \overset{1\mathfrak{p}}{\eta}(x) \coloneqq \{\!\{x\}\!\} \\ \overset{1\mathfrak{p}}{\mu} : \mathfrak{p}(A) \nearrow \wp(A) & & \overset{1\mathfrak{p}}{\mu}(\mathfrak{X}) \coloneqq \{x \mid x \in \mathfrak{X}\} \end{array}$$

Finally, we redefine abstract configurations $(\varsigma^\sharp \in \Sigma^\sharp)$ to use constructive finite sets:

$$\varsigma^\sharp \in \Sigma^\sharp \coloneqq \mathtt{env}^\sharp \times \mathfrak{p}(\mathtt{cexp})$$

In this new setting for abstract configurations, the constructive Galois connection for concrete configurations $(\varsigma \in \Sigma)$ is:

$$\Sigma \underset{\overset{r\times 1\mathfrak{p}}{\eta}}{\overset{\overset{r\times 1}{\mu}}{\leftrightarrows}} \Sigma^\sharp \qquad \begin{array}{l} \overset{r\times 1\mathfrak{p}}{\eta} : \Sigma \to \Sigma^\sharp \\ \overset{r\times 1\mathfrak{p}}{\mu} : \Sigma^\sharp \nearrow \wp(\Sigma) \end{array}$$

$$\begin{array}{rcl} \overset{r\times 1\mathfrak{p}}{\eta}(\rho, ce) & \coloneqq & \langle \eta^r(\rho), \{\!\{ce\}\!\} \rangle \\ \overset{r\times 1\mathfrak{p}}{\mu}(\rho^\sharp, CE) & \coloneqq & \{\langle \rho, ce \rangle \mid \rho \in \mu^r(\rho^\sharp) \wedge cd \in CE\} \end{array}$$

Using constructive finite sets and this new definition for abstract configurations, we will perform the same calculation as before, but entirely within the constructive

Galois connection framework, and in abstraction-directed form.

THE CALCULATION We show the calculation for the abstract interpretation of conditional expressions using constructive Galois connections in figures 4.17 and 4.18. Steps 1–3 unfold semantic function and relation definitions; Step 4 applies commutativity of set union; Step 5 pushes abstraction through the set comprehension; Step 6 introduces adjacent concretization and abstraction functions, justified by Galois connection extensiveness (an explicit loss in precision); Step 7 applies the constructive Galois connection correspondence; Step 8 applies a correct abstract interpreter for boolean expressions; Step 9 pulls abstraction out of the set comprehension; Step 10 collapses adjacent abstraction and concretization functions; and Step 11 declares the final state of the calculation as the definition of the algorithm.

What this calculation shows is that constructive Galois connections support manipulating multivalued abstractions and algorithms, via an explicit finite set construction, which carries algorithmic content in a constructive logic setting. What classically was just a powerset with finite elements becomes an explicit finite set, and what classically was an undecidable specification of potentially infinite elements remains a powerset. Supporting relational abstraction can be done in this way as well, for example a relational abstraction for environments would have the shape of:

$$\overset{rel}{\eta^r} \; : \; \mathfrak{p}(env) \nearrow env^\sharp \qquad\qquad \overset{rel}{\mu^r} \; : \; env^\sharp \nearrow \wp(\mathfrak{p}(env))$$

$$
\begin{array}{ll}
 & \lfloor \eta^{\Sigma} \rfloor^{*}(\mathcal{C}[\texttt{if}\ be\ \texttt{then}\ ce_1\ \texttt{else}\ ce_2]^{*}(\mu^{r}(\rho^{\sharp}))) \\
(1) & = \ \wr \text{ defn. of } \mathcal{C}[\texttt{if}\ be\ \texttt{then}\ ce_1\ \texttt{else}\ ce_2] \int \\
 & \lfloor \eta^{\Sigma} \rfloor^{*}(\bigcup_{\rho \in \mu^{r}(\rho^{\sharp})} \{\langle \rho', ce \rangle \mid \langle \rho, \texttt{if}\ be\ \texttt{then}\ ce_1\ \texttt{else}\ ce_2 \rangle \mapsto^{c} \langle \rho', ce \rangle\}) \\
(2) & = \ \wr \text{ defn. of } \langle \rho, \texttt{if}\ be\ \texttt{then}\ ce_1\ \texttt{else}\ ce_2 \rangle \mapsto^{c} \langle \rho', ce' \rangle \int \\
 & \lfloor \eta^{\Sigma} \rfloor^{*}(\bigcup_{\rho \in \mu^{r}(\rho^{\sharp})} \{\langle \rho, ce_1 \rangle \mid \rho \vdash be \Downarrow^{b} true\} \cup \{\langle \rho, ce_2 \rangle \mid \rho \vdash be \Downarrow^{b} false\}) \\
(3) & = \ \wr \text{ defn. of } \rho \vdash be \Downarrow^{b} b \int \\
 & \lfloor \eta^{\Sigma} \rfloor^{*}(\bigcup_{\rho \in \mu^{r}(\rho^{\sharp})} \{\langle \rho, ce_1 \rangle \mid true = \mathcal{B}[be](\rho)\} \cup \{\langle \rho, ce_2 \rangle \mid false = \mathcal{B}[be](\rho)\}) \\
(4) & = \ \wr \text{ set equality (union commutativity) } \int \\
 & \lfloor \eta^{\Sigma} \rfloor^{*} \left(\bigcup \left\{ \begin{array}{l} \bigcup_{\rho \in \mu^{r}(\rho^{\sharp})} \{\langle \rho, ce_1 \rangle \mid true = \mathcal{B}[be](\rho)\} \\ \bigcup_{\rho \in \mu^{r}(\rho^{\sharp})} \{\langle \rho, ce_2 \rangle \mid false = \mathcal{B}[be](\rho)\} \end{array} \right. \right) \\
(5) & = \ \wr \text{ set equality } \int \\
 & \bigcup \left\{ \begin{array}{l} \bigcup_{\rho \in \mu^{r}(\rho^{\sharp})} \{\langle \eta^{r}(\rho), \{\!\!\{ce_1\}\!\!\}\rangle \mid true = \mathcal{B}[be](\rho)\} \\ \bigcup_{\rho \in \mu^{r}(\rho^{\sharp})} \{\langle \eta^{r}(\rho), \{\!\!\{ce_2\}\!\!\}\rangle \mid false = \mathcal{B}[be](\rho)\} \end{array} \right. \\
(6) & \sqsubseteq \ \wr \ \mu^{b} \circledast \lfloor \eta^{b} \rfloor \text{ extensive } (ret \sqsubseteq \mu^{b} \circledast \lfloor \eta^{b} \rfloor) \int \\
 & \bigcup \left\{ \begin{array}{l} \bigcup_{\rho \in \mu^{r}(\rho^{\sharp})} \{\langle \eta^{r}(\rho), \{\!\!\{ce_1\}\!\!\}\rangle \mid true \in \mu^{b}(\eta^{b}(\mathcal{B}[be](\rho)))\} \\ \bigcup_{\rho \in \mu^{r}(\rho^{\sharp})} \{\langle \eta^{r}(\rho), \{\!\!\{ce_2\}\!\!\}\rangle \mid false \in \mu^{b}(\eta^{b}(\mathcal{B}[be](\rho)))\} \end{array} \right. \\
 & \ldots
\end{array}
$$

Figure 4.17: Conditional Expressions Constructive Calculation

$$
\begin{array}{ll}
 & \ldots \\
\textit{(7)} & = \ \wr \text{ constructive GC correspondence } (b \in \mu^b(b^\sharp) \Leftrightarrow \eta^b(b) \sqsubseteq b^\sharp) \ \int \\
 & \bigcup \left\{ \begin{array}{l} \bigcup\limits_{\rho \in \mu^r(\rho^\sharp)} \{\langle \eta^r(\rho), \{\!|ce_1|\!\}\rangle \mid true \sqsubseteq \eta^b(\mathcal{B}[be](\rho))\} \\ \bigcup\limits_{\rho \in \mu^r(\rho^\sharp)} \{\langle \eta^r(\rho), \{\!|ce_2|\!\}\rangle \mid false \sqsubseteq \eta^b(\mathcal{B}[be](\rho))\} \end{array} \right. \\
\textit{(8)} & \sqsubseteq \ \wr \ \mathcal{B}^\sharp[\_] \text{ correct } (\eta^b \circ \mathcal{B}[be] \sqsubseteq \mathcal{B}^\sharp[be] \circ \eta^r) \ \int \\
 & \bigcup \left\{ \begin{array}{l} \bigcup\limits_{\rho \in \mu^r(\rho^\sharp)} \{\langle \eta^r(\rho), \{\!|ce_1|\!\}\rangle \mid true \sqsubseteq \mathcal{B}^\sharp[be](\eta^r(\rho))\} \\ \bigcup\limits_{\rho \in \mu^r(\rho^\sharp)} \{\langle \eta^r(\rho), \{\!|ce_2|\!\}\rangle \mid false \sqsubseteq \mathcal{B}^\sharp[be](\eta^r(\rho))\} \end{array} \right. \\
\textit{(9)} & = \ \wr \text{ set equality } \int \\
 & \bigcup \left\{ \begin{array}{l} \bigcup\limits_{\rho^{\sharp\prime} \in \lfloor \eta^r \rfloor^* \mu^r(\rho^\sharp)} \{\langle \rho^{\sharp\prime}, \{\!|ce_1|\!\}\rangle \mid true \sqsubseteq \mathcal{B}^\sharp[be](\rho^{\sharp\prime})\} \\ \bigcup\limits_{\rho^{\sharp\prime} \in \lfloor \eta^r \rfloor^* \mu^r(\rho^\sharp)} \{\langle \rho^{\sharp\prime}, \{\!|ce_2|\!\}\rangle \mid false \sqsubseteq \mathcal{B}^\sharp[be](\rho^{\sharp\prime})\} \end{array} \right. \\
\textit{(10)} & \sqsubseteq \ \wr \ \lfloor \eta^r \rfloor \circledast \mu^r \text{ reductive } (\lfloor \eta^r \rfloor \circledast \mu^b \sqsubseteq ret) \ \int \\
 & \left\{ \left\langle \rho^\sharp, \bigcup \left\{ \begin{array}{lll} \{\!|ce_1|\!\} & if & true \sqsubseteq \mathcal{B}^\sharp[be](\rho^\sharp) \\ \{\!|ce_2|\!\} & if & false \sqsubseteq \mathcal{B}^\sharp[be](\rho^\sharp) \end{array} \right. \right\rangle \right\} \\
\textit{(11)} & \triangleq \ \wr \text{ by } \mathcal{C}^\sharp[\texttt{if } be \texttt{ then } ce_1 \texttt{ else } ce_2](\rho^\sharp) \coloneqq \left\langle \rho^\sharp, \bigcup \left\{ \begin{array}{lll} \{\!|ce_1|\!\} & if & true \sqsubseteq \mathcal{B}^\sharp[be](\rho^\sharp) \\ \{\!|ce_2|\!\} & if & false \sqsubseteq \mathcal{B}^\sharp[be](\rho^\sharp) \end{array} \right. \right\rangle \ \int \\
 & \lfloor \mathcal{C}^\sharp[\texttt{if } be \texttt{ then } ce_1 \texttt{ else } ce_2] \rfloor (\rho^\sharp)
\end{array}
$$

Figure 4.18: Conditional Expressions Constructive Calculation (Cont.)

4.11 Related Work

This work connects two long strands of research: abstract interpretation *via* Galois connections and mechanized verification *via* dependently typed functional programming. The former is founded on the pioneering work of Cousot and Cousot [1977, 1979]; the latter on that of Martin-Löf [1984], embodied in Norell's Agda [Norell, 2007]. Our key technical insight is to use a monadic structure for Galois connections, following the example of Moggi [1989] for the λ-calculus.

Calculational Abstract Interpretation Cousot describes calculational abstract interpretation by example in his lecture notes [2005] and monograph [1999], and Cousot and Cousot recently introduced a unifying calculus for Galois connections [2014]. Our work mechanizes Cousot's calculations and provides a foundation for mechanizing other instances of calculational abstract interpretation (*e.g.*, [Midtgaard and Jensen, 2008, Sergey et al., 2012]). We expect our work to have applications to the mechanization of calculational program design [Bird and de Moor, 1996, Bird, 1990] by employing only Galois *retractions*, *i.e.* $\alpha \circ \gamma$ is an identity [Cousot and Cousot, 2014]. There is prior work on mechanized program calculation [Tesson et al., 2011], but it is not based on abstract interpretation.

Verified Static Analyzers Verified abstract interpretation has shown many promising results [Barthe et al., 2007, Blazy et al., 2013, Cachera and Pichardie, 2010, Pichardie, 2005], scaling up to large-scale real-world static analyzers [Jourdan et al., 2015]. However, mechanized abstract interpretation has yet to benefit from

the Galois connection framework. Until now, approaches use classical axioms or "γ-only" encodings of soundness and (sometimes) completeness. Our techniques for mechanizing Galois connections should complement these approaches.

GALCULATOR The Galculator [Silva and Oliveira, 2008] is a proof assistant founded on an algebra of Galois connections. This tool is similar to ours in that it mechanically verifies Galois connection calculations. Our approach is more general, supporting arbitrary set-theoretic reasoning and embedded within a general purpose proof assistant, however their approach is fully automated for the small set of derivations which reside within their supported theory.

DEDUCTIVE SYNTHESIS Fiat [Delaware et al., 2015] is a library for the Coq proof assistant which supports semi-automated synthesis of programs as refinements of their specifications. Fiat uses the same powerset type and monad as we do, and their "deductive synthesis" process similarly derives correct-by-construction programs by calculus. Fiat derivations start with a user-defined specification and calculate towards an *under*-approximation ($\sqsupseteq$), whereas calculational abstract interpretation starts with an optimal specification and calculates towards an *over*-approximation ($\sqsubseteq$). It should be possible to generalize their framework to use partial orders to recover aspects of our work, or to invert the lattice used in our abstract interpretation framework to recover aspects of theirs. A notable difference in approach is that Fiat makes heavy use of Coq's tactic programming language to automate rewrites inside respectful contexts, whereas our system provides no interactive proof automation

and each calculational step must be notated explicitly.

Monadic Abstract Interpretation Monads in abstract interpretation have recently been applied to good effect for modularity [Darais et al., 2015, Sergey et al., 2013]. However, that work uses monads to structure the semantics, not the Galois connections and proofs.

Future Directions Now that we have established a foundation for constructive Galois connection calculation, we see value in verifying larger derivations (*e.g.*, [Midtgaard and Jensen, 2008, Sergey et al., 2012]). Furthermore we would like to explore whether or not our techniques have any benefit in the space of general-purpose program calculations *à la* Bird.

Currently our framework requires the user to justify every detail of the program calculation, including monotonicity proofs and proof scoping for rewrites inside monotonic contexts. We imagine much of this can be automated, requiring the user to only provide the interesting parts of the proof, à la Fiat [Delaware et al., 2015]. Our experience has been that even Coq's tactic system slows down considerably when automating all of these details, and we foresee using proof by reflection in either Coq (*e.g.*, Rtac [Malecha and Bengtson, 2016]) or Agda to automate these proofs in a way that maintains proof-checker performance.

There have been recent developments on compositional abstract interpretation frameworks [Darais et al., 2015] where abstract interpreters and their proofs of soundness are systematically derived side-by-side. That framework relies on correct-

ness properties transported by *Galois transformers*, which we posit would benefit from mechanization since they hold both computational and specification content.

4.12 Conclusions

This chapter realizes the vision of mechanized and constructive Galois connections foreshadowed by Cousot [1999, p. 85], giving the first mechanically verified proof by calculational abstract interpretation; once for his generic static analyzer and once for the semantics of gradual typing. Our proofs by calculus closely follow the originals. The primary discrepancy is the use of monads to isolate *specification effects*. By maintaining this discipline, we are able to verify calculations by Galois connections *and* extract computational content from pure results. The resulting artifacts are correct-by-verified-construction, thereby avoiding known bugs in the original.[2]

[2] http://www.di.ens.fr/~cousot/aisoftware/Marktoberdorf98/Bug_History

Chapter 5: Galois Transformers

5.1 Introduction

Traditional practice in program analysis via abstract interpretation is to fix a language (as a concrete semantics) and an abstraction (as an abstraction map, concretization map or Galois connection) before constructing a static analyzer that is sound with respect to both the abstraction and the concrete semantics. Thus, each pairing of abstraction and semantics requires a one-off manual derivation of the static analyzer and construction of its proof of soundness.

Work has focused on endowing abstractions with knobs, levers, and dials to tune precision and compute efficiently. These parameters come with overloaded meanings such as object, context, path and heap sensitivities, or some combination thereof. These efforts develop families of analyses *for a specific language* and prove the framework sound.

But this framework approach suffers from many of the same drawbacks as the one-off analyzers. They are language-specific, preventing reuse of concepts across languages, and require similar re-implementations and soundness proofs. This process is still manual, tedious, difficult and error-prone. And, changes to the structure of the parameter-space require a completely new proof of soundness. And, it prevents

fruitful insights and results developed in one paradigm from being applied to others, *e.g.*, functional to object-oriented and *vice versa*.

We propose an automated alternative to structuring and implementing program analysis. Inspired by Liang et al.'s *Monad Transformers and Modular Interpreters* [1995], we propose to start with concrete interpreters written in a specific monadic style. Changing the monad will transform the concrete interpreter into an abstract interpreter. As we show, classical program abstractions can be embodied as language-independent monads. Moreover, these abstractions can be written as monad *transformers*, thereby allowing their composition to achieve new forms of analysis. We show that these monad transformers obey the properties of *Galois connections* [Cousot and Cousot, 1979] and introduce the concept of a *Galois transformer*, a monad transformer which transports Galois connection properties.

Most significantly, Galois transformers are proven sound once and for all. Abstract interpreters, which take the form of monad transformer stacks coupled with a monadic interpreter, inherit the soundness properties of each element in the stack. This approach enables reuse of abstractions across languages and lays the foundation for a modular metatheory of program analysis.

SETUP We describe a simple programming language and a garbage-collecting allocating semantics as the starting point of analysis design (§ 5.2). We then briefly discuss three types of path and flow sensitivity and their corresponding variations in analysis precision (§ 5.3).

Monadic Abstract Interpreters We develop an abstract interpreter for our example language as a monadic function with parameters (§ 5.2 and 5.5), one of which is a monadic effect interface combining state and nondeterminism effects (§ 5.4.1). These monadic effects—state and nondeterminism—encode arbitrary relational small-step state-machine semantics and correspond to state-machine components and relational nondeterminism, respectively.

Interpreters written in this style are reasoned about using various laws, including monadic effect laws, and are verified correct independent of any particular choice of parameters. Likewise, choices for these parameters are proven correct in isolation from their instantiation. When instantiated, our generic interpreter recovers the concrete semantics and a family of abstract interpreters with variations in abstract domain, abstract garbage collection, call-site sensitivity, object sensitivity, and path and flow sensitivity (§ 5.6). Furthermore, each derived abstract interpreter is proven correct by construction through a reusable, semantics independent proof framework (§ 5.8).

Isolating Path and Flow Sensitivity We give specific monads for instantiating the interpreter from Section 5.5 to path-sensitive, flow-sensitive and flow-insensitive analyses (§ 5.7). This leads to an isolated understanding of path and flow sensitivity as mere variations in the monad used for execution. Furthermore, these monads are language independent, allowing one to reuse the same path and flow sensitivity machinery for any language of interest, and compose seamlessly with other analysis parameters.

Galois Transformers To ease the construction of monads for building abstract interpreters and their proofs of correctness, we develop a framework of *Galois transformers* (§ 5.8). Galois transformers are an extension of monad transformers which transport Galois connection properties (§ 5.8.4). The Galois transformer framework allows us to both execute and justify the correctness of an abstract interpreter piecewise for each transformer. Galois transformers are language independent and they are proven correct once and for all in isolation from a particular semantics.

Implementation We implement our technique as a Haskell library and example client analysis (§ 5.9). Developers are able to reuse our language-independent framework for prototyping the design space of analysis features for their language of choice. Our implementation is publicly available on Hackage[1], Haskell's package manager.

Contributions We make the following contributions:

- A methodology for constructing monadic abstract interpreters based on *monadic effects*.
- A compositional, language-independent framework for constructing monads with varying analysis properties based on *monad transformers*.
- A compositional, language-independent proof framework for constructing Galois connections and end-to-end correctness proofs based on *Galois transformers*, an extension of monad transformers which transports Galois connection properties.

[1] http://hackage.haskell.org/package/maam

- Two new general purpose monad transformers for nondeterminism which are not present in any previous work on monad transformers (even outside static analysis literature). Although applicable to settings other than static analysis, these two transformers give rise naturally to variations in path and flow sensitivity when applied to abstract interpreters.

- An isolated understanding of path and flow sensitivity in analysis as properties of the interpreter monad, which we develop independently of other analysis features.

Collectively, these contributions make progress toward a reusable metatheory for program analysis.

5.2 Semantics

To demonstrate our framework we design an abstract interpreter for $\lambda\texttt{IF}$, a simple applied lambda calculus shown in Figure 5.1. $\lambda\texttt{IF}$ extends traditional lambda calculus with integers, addition, subtraction and conditionals. We write @ as explicit abstract syntax for function application. The state-space Σ for $\lambda\texttt{IF}$ makes allocation explicit using two separate stores for values (*Store*) and for the stack (*KStore*).

Guided by the syntax and semantics of $\lambda\texttt{IF}$ we develop interpretation parameters in Section 5.4, a monadic interpreter in Section 5.5, and both concrete and abstract instantiations for the interpretation parameters in Section 5.6. The variations in path and flow sensitivity developed in sections 5.7 and 5.8 are independent of this (or any other) semantics.

$$
\begin{array}{rrcl}
i \in & \mathbb{Z} & & \\
x \in & \mathit{Var} & & \\
a \in & \mathit{Atom} & ::= & i \mid x \mid \lambda x.e \\
\oplus \in & \mathit{IOp} & ::= & + \mid - \\
\odot \in & \mathit{Op} & ::= & \oplus \mid @ \\
e \in & \mathit{Exp} & ::= & a \mid e \odot e \mid \texttt{if0}(e)\{e\}\{e\} \\
 & & & \\
\tau \in & \mathit{Time} & ::= & \mathbb{Z} \\
l \in & \mathit{Addr} & ::= & \mathit{Var} \times \mathit{Time} \\
\rho \in & \mathit{Env} & ::= & \mathit{Var} \rightharpoonup \mathit{Addr} \\
\sigma \in & \mathit{Store} & ::= & \mathit{Addr} \rightharpoonup \mathit{Val} \\
c \in & \mathit{Clo} & ::= & \langle \lambda x.e, \rho \rangle \\
v \in & \mathit{Val} & ::= & i \mid c \\
\kappa l \in & \mathit{KAddr} & ::= & \mathit{Time} \\
\kappa\sigma \in & \mathit{KStore} & ::= & \mathit{KAddr} \rightharpoonup \mathit{Frame} \times \mathit{KAddr} \\
fr \in & \mathit{Frame} & ::= & \langle \square \odot e, \rho \rangle \mid \langle v \odot \square \rangle \mid \langle \texttt{if0}(\square)\{e\}\{e\}, \rho \rangle \\
\varsigma \in & \Sigma & ::= & \langle e, \rho, \sigma, \kappa l, \kappa\sigma, \tau \rangle
\end{array}
$$

Figure 5.1: λIF Syntax and Concrete State Space

We define semantics for atomic expressions and primitive operators denotationally with $A[\![\_]\!]$ and $\delta[\![\_]\!]$, and to compound expressions relationally with $\_ \rightsquigarrow \_$, shown in Figure 5.2.

Our abstract interpreter supports abstract garbage collection [Might and Shivers, 2006a], the concrete analogue of which is just standard garbage collection. We include abstract garbage collection for two reasons. First, it is one of the few techniques that results in both performance *and* precision improvements for abstract interpreters. Second, we will systematically recover concrete and abstract garbage collectors with varying path and flow sensitivities through a single monadic garbage collector, an axis of generality novel in this work.

We show the garbage collected semantics in Figure 5.3, as well as a final collecting semantics *collect*, which will serve as the starting point for abstraction. The concrete, garbage-collected collecting semantics *collect* and a sound static analyzer will both be recovered from instantiations of a generic monadic interpreter in Section 5.6.

The garbage collected semantics $\_ \rightsquigarrow^{gc} \_$ is defined with reachability functions KR and R which define transitively reachable addresses. We write $\mu X.f(X)$ as the least-fixed-point of the function f. R is defined in terms of $R\text{-}Frm$ and $R\text{-}Val$, which define the immediately reachable locations from a frame and value respectively. We omit the definition of FV, which is the standard recursive definition for computing free variables of an expression.

$$A[\![\_]\!] : \mathit{Atom} \to \mathit{Env} \times \mathit{Store} \rightharpoonup \mathit{Val}$$
$$A[\![i]\!](\rho, \sigma) \coloneqq i$$
$$A[\![x]\!](\rho, \sigma) \coloneqq \sigma(\rho(x))$$
$$A[\![\lambda x.e]\!](\rho, \sigma) \coloneqq \langle \lambda x.e, \rho \rangle$$

$$\delta[\![\_]\!] : \mathit{IOp} \to \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$$
$$\delta[\![+]\!](i_1, i_2) \coloneqq i_1 + i_2$$
$$\delta[\![-]\!](i_1, i_2) \coloneqq i_1 - i_2$$

$$\_ \rightsquigarrow \_ : \wp(\Sigma \times \Sigma)$$

$$\langle e_1 \odot e_2, \rho, \sigma, \kappa l, \kappa\sigma, \tau \rangle \rightsquigarrow \langle e_1, \rho, \sigma, \tau, \kappa\sigma', \tau + 1 \rangle \texttt{ where}$$
$$\kappa\sigma' \coloneqq \kappa\sigma[\tau \mapsto \langle \langle \square \odot e_2, \rho \rangle, \kappa l \rangle]$$

$$\langle \texttt{if0}(e_1)\{e_2\}\{e_3\}, \rho, \sigma, \kappa l, \kappa\sigma, \tau \rangle \rightsquigarrow \langle e_1, \rho, \sigma, \tau, \kappa\sigma', \tau + 1 \rangle \texttt{ where}$$
$$\kappa\sigma' \coloneqq \kappa\sigma[\tau \mapsto \langle \langle \texttt{if0}(\square)\{e_2\}\{e_3\}, \rho \rangle, \kappa l \rangle]$$

$$\langle a, \rho, \sigma, \kappa l, \kappa\sigma, \tau \rangle \rightsquigarrow \langle e, \rho', \sigma, \tau, \kappa\sigma', \tau + 1 \rangle \texttt{ where}$$
$$\langle \langle \square \odot e, \rho' \rangle, \kappa l' \rangle \coloneqq \kappa\sigma(\kappa l)$$
$$\kappa\sigma' \coloneqq \kappa\sigma[\tau \mapsto \langle \langle A[\![a]\!](\rho, \sigma) \odot \square \rangle, \kappa l' \rangle]$$

$$\langle a, \rho, \sigma, \kappa l, \kappa\sigma, \tau \rangle \rightsquigarrow \langle e, \rho'', \sigma', \kappa l', \kappa\sigma, \tau + 1 \rangle \texttt{ where}$$
$$\langle \langle \langle \lambda x.e, \rho' \rangle @ \square \rangle, \kappa l' \rangle \coloneqq \kappa\sigma(\kappa l)$$
$$\rho'' \coloneqq \rho'[x \mapsto \langle x, \tau \rangle]$$
$$\sigma' \coloneqq \sigma[\langle x, \tau \rangle \mapsto A[\![a]\!](\rho, \sigma)]$$

$$\langle i_2, \rho, \sigma, \kappa l, \kappa\sigma, \tau \rangle \rightsquigarrow \langle i, \rho, \sigma, \kappa l', \kappa\sigma, \tau + 1 \rangle \texttt{ where}$$
$$\langle \langle i_1 \oplus \square \rangle, \kappa l' \rangle \coloneqq \kappa\sigma(\kappa l)$$
$$i \coloneqq \delta[\![\oplus]\!](i_1, i_2)$$

$$\langle i, \rho, \sigma, \kappa l, \kappa\sigma, \tau \rangle \rightsquigarrow \langle e, \rho', \sigma, \kappa l', \kappa\sigma, \tau + 1 \rangle \texttt{ where}$$
$$\langle \langle \texttt{if0}(\square)\{e_1\}\{e_2\}, \rho' \rangle, \kappa l' \rangle \coloneqq \kappa\sigma(\kappa l)$$
$$e \coloneqq e_1 \texttt{ when } i = 0 \,;\, e_2 \texttt{ when } i \neq 0$$

Figure 5.2: Concrete Semantics

$$
\begin{array}{l}
\_ \rightsquigarrow^{gc} \_ \;:\; \wp(\Sigma \times \Sigma) \\
\varsigma \rightsquigarrow^{gc} \varsigma' \;\texttt{where}\; \varsigma \rightsquigarrow \varsigma' \\
\langle e, \rho, \sigma, \kappa l, \kappa\sigma, \tau \rangle \rightsquigarrow^{gc} \langle e, \rho, \sigma', \kappa l, \kappa\sigma', \tau \rangle \;\texttt{where} \\
\quad \kappa\sigma' \;\coloneqq\; \{\kappa l \mapsto \kappa\sigma(\kappa l) \mid \kappa l \in KR(\kappa l, \kappa\sigma)\} \\
\quad \sigma' \;\coloneqq\; \{l \mapsto \sigma(l) \mid l \in R(e, \rho, \sigma, \kappa l, \kappa\sigma)\} \\
\\
KR \;:\; KAddr \times KStore \to \wp(KAddr) \\
KR(\kappa l, \kappa\sigma) \;\coloneqq\; \mu X. \\
\quad X \cup \{\kappa l\} \cup \{\pi_2(\kappa\sigma(\kappa l)) \mid \kappa l \in X\} \\
R \;:\; Exp \times Env \times Store \times KAddr \times KStore \to \wp(Addr) \\
R(e, \rho, \sigma, \kappa l, \kappa\sigma) \;\coloneqq\; \mu X. \\
\quad X \cup \{\rho(x) \mid x \in FV(e)\} \\
\quad \cup \{l \mid l \in R\text{-}Frm(\pi_1(\kappa\sigma(\kappa l)))\;;\; \kappa l \in KR(\kappa l, \kappa\sigma)\} \\
\quad \cup \{l' \mid l' \in R\text{-}Val(\sigma(l))\;;\; l \in X\} \\
\\
R\text{-}Frm \;:\; Frame \to \wp(Addr) \\
R\text{-}Frm(\langle \square \odot e, \rho \rangle) \;\coloneqq\; \{\rho(x) \mid x \in FV(e)\} \\
R\text{-}Frm(\langle v \odot \square \rangle) \;\coloneqq\; R\text{-}Val(v) \\
R\text{-}Frm(\langle \texttt{if0}(\square)\{e_2\}\{e_3\}, \rho \rangle) \;\coloneqq\; \{\rho(x) \mid x \in FV(e_1) \cup FV(e_2)\} \\
R\text{-}Val \in Val \to \wp(Addr) \\
R\text{-}Val(i) \;\coloneqq\; \{\} \\
R\text{-}Val(\langle \lambda x.e, \rho \rangle) \;\coloneqq\; \{\rho(y) \mid y \in FV(\lambda x.e)\} \\
\\
collect \;:\; \wp(\Sigma) \\
collect \;\coloneqq\; \mu X. X \cup \{\varsigma_0\} \cup \{\varsigma' \mid \varsigma \rightsquigarrow^{gc} \varsigma'\;;\; \varsigma \in X\} \;\texttt{where} \\
\quad \varsigma_0 \;\coloneqq\; \langle e_0, \bot, \bot, 0, \bot, 1 \rangle
\end{array}
$$

Figure 5.3: Garbage Collected Collecting Semantics

5.3 Path and Flow Sensitivity in Analysis

We identify three specific variants of path and flow sensitivity in analysis: path-sensitive, flow-sensitive and flow-insensitive. Our framework exposes the essence of path and flow sensitivity through a monadic effect interface in Section 5.4, and we recover each of these variations through specific monad instances in sections 5.7 and 5.8.

Consider a combination of if-statements in our example language λIF (extended with let-bindings) where an analysis cannot determine the value of N:

```
(1)  let x :=
(2)    if0(N){
(3)      if0(N){1}{2} }{
(4)      if0(N){3}{4} } in
(5)  let y := if0(N){5}{6} in
(6)  exit(x, y)
```

Path-Sensitive A path-sensitive analysis tracks both data and control flow precisely. At lines 3 and 4 the analysis considers separate worlds:

$$3: \ \{N = 0\} \qquad 4: \ \{N \neq 0\}$$

At Line 5 the analysis continues in two separate, precise worlds:

$$5: \ \{N = 0, x = 1\} \ \{N \neq 0, x = 4\}$$

At Line 6 the analysis correctly correlates x and y:

$$6: \ \{N = 0, x = 1, y = 5\} \ \{N \neq 0, x = 4, y = 6\}$$

FLOW-SENSITIVE A flow-sensitive analysis collects a *single* set of facts for each variable *at each program point*. At lines 3 and 4, the analysis considers separate worlds:

$$3: \ \{N = 0\} \qquad 4: \ \{N \neq 0\}$$

Each nested if-statement then evaluates only one side of the branch, resulting in values 1 and 4. At Line 5 the analysis is only allowed one set of facts, so it must merge the possible values that x and N could take:

$$5: \ \{N \in \mathbb{Z}, x \in \{1, 4\}\}$$

The analysis then explores both branches at Line 5 resulting in no correlation between values for x and y at Line 6:

$$6: \ \{N \in \mathbb{Z}, x \in \{1, 4\}, y \in \{5, 6\}\}$$

FLOW-INSENSITIVE A flow-insensitive analysis collects a *single* set of facts about each variable which must hold true *for the entire program*. Because the value of N is unknown at *some* point in the program, the value of x must consider both branches of the nested if-statement. This results in the global set of facts giving four values

to x:

$$\textit{1–6}: \quad \{N \in \mathbb{Z}, x \in \{1,2,3,4\}, y \in \{5,6\}\}$$

5.4 Analysis Parameters

Before constructing the abstract interpreter we first design its parameters. The interpreter, which we develop in Section 5.5, will be designed such that variations in these parameters will recover both concrete and a family of abstract interpreters, which we show in Section 5.6. To do this we extend the ideas developed in Van Horn and Might [2010] with a new parameter for path and flow sensitivity: the interpreter monad.

There will be three parameters to our abstract interpreter:

1. The monad, novel in this work, which captures control effects and gives rise to path and flow sensitivity.

2. The abstract domain, which captures the abstraction of values like integers or datatypes.

3. The abstraction for time, which captures call-site and object sensitivities.

We place each of these parameters behind an abstract interface and leave their implementations opaque when defining the monadic interpreter in Section 5.5. Each parameter comes with laws which can be used to reason about the generic interpreter independent of a particular instantiation. Likewise, an instantiation of the interpreter

need only justify that each parameter meets its local interface, which we justify in isolation from the generic interpreter.

5.4.1 The Analysis Monad

The monad for the interpreter captures the *effects* of interpretation. There are two effects in the interpreter: state and nondeterminism. The state effect will mediate how the interpreter interacts with state cells in the state space: *Env*, *Store*, *KAddr*, *KStore* and *Time*. The nondeterminism effect will mediate branching in the execution of the interpreter. Path and flow sensitivity will be recovered by altering how these effects interact in a particular choice of monad.

We use monadic state and nondeterminism effects to abstract over arbitrary relational small-step state-machine semantics. State effects correspond to the components of the state-machine and nondeterminism effects correspond to potential nondeterminism in the relation's definition.

We briefly review monad, state and nondeterminism operators and their laws. For a more details see Gibbons and Hinze [2011], Liang et al. [1995], Moggi [1989].

Monad Operators A type operator m is a monad if it supports *bind*, a sequencing operator, and its unit *return*:

$$\begin{aligned} m &: \mathit{Type} \to \mathit{Type} \\ \mathit{return} &: \forall A.A \to m(A) \\ \mathit{bind} &: \forall AB.m(A) \to (A \to m(B)) \to m(B) \end{aligned}$$

and obeys left unit, right unit and associativity laws.

We use semicolon notation for *bind* (*e.g.*, $x \leftarrow X\ ;\ k(x)$ is sugar for $bind(X)(k)$) and we replace semicolons with line breaks headed by `do` for multiline monadic definitions.

State Effect A type operator m supports the monadic state effect for a type s if it supports *get* and *put* operations over s:

$$\begin{aligned} s &: \mathit{Type} & \mathit{get} &: m(s) \\ m &: \mathit{Type} \to \mathit{Type} & \mathit{put} &: s \to m(\mathit{unit}) \end{aligned}$$

and obeys get-get, get-put, put-get and put-put laws [Gibbons and Hinze, 2011].

Nondeterminism Effect A type operator m supports the monadic nondeterminism effect if it supports an alternation operator $\boxplus$ and its unit *mzero*:

$$\begin{aligned} m &: \mathit{Type} \to \mathit{Type} \\ \_\boxplus\_ &: \forall A.m(A) \times m(A) \to m(A) \\ \mathit{mzero} &: \forall A.m(A) \end{aligned}$$

The type $m(A)$ must have a join-semilattice structure, *mzero* must be a zero for *bind*, and *bind* must distribute through $\boxplus$.

The interpreter in Section 5.5 will be defined generic to a monad which supports monad operators, state effects and nondeterminism effects. As a consequence, we do not reference an explicit configuration ς or collections of results; instead we interact with an interface of state and nondeterminism effects. This level of indirection will be exploited in Section 5.7, where different monads will meet the same effect interface but yield different analysis properties.

5.4.2 The Abstract Domain

To expose the abstract domain we parameterize over *Val*, introduction and elimination forms for *Val*, and the denotation for primitive operators $\delta[\![\_]\!]$.

Val must be a join-semilattice with $\sqcup$ and its unit $\bot$:

$$\bot \ : \ \mathit{Val} \qquad\qquad \_ \sqcup \_ \ : \ \mathit{Val} \times \mathit{Val} \to \mathit{Val}$$

and respect the usual join-semilattice laws. *Val* must be a join-semilattice so it can be merged in updates to *Store* to preserve soundness.

Val must also support introduction and elimination between finite sets of concrete values $\mathbb{Z}$ and *Clo*:

$$\begin{array}{ll} \mathit{int\text{-}I} \ : \ \mathbb{Z} \to \mathit{Val} & \mathit{if0\text{-}E} \ : \ \mathit{Val} \to \wp(\mathit{Bool}) \\ \mathit{clo\text{-}I} \ : \ \mathit{Clo} \to \mathit{Val} & \mathit{clo\text{-}E} \ : \ \mathit{Val} \to \wp(\mathit{Clo}) \end{array}$$

Introduction functions inject concrete values into abstract values. Elimination functions project abstract values into a *finite* set of concrete observations. For example, we do not require that abstract values support elimination to integers, only to finite observation of comparison with zero. The laws for the introduction and elimination functions induce a Galois connection between $\wp(\mathbb{Z})$ and *Val*:

$$\begin{array}{rcl} \{\mathit{true}\} & \subseteq & \mathit{if0\text{-}E}(\mathit{int\text{-}I}(i)) \ \texttt{if} \ i = 0 \\ \{\mathit{false}\} & \subseteq & \mathit{if0\text{-}E}(\mathit{int\text{-}I}(i)) \ \texttt{if} \ i \neq 0 \\ \displaystyle\bigsqcup_{\substack{b \in \mathit{if0\text{-}E}(v) \\ i \in \theta(b)}} \mathit{int\text{-}I}(i) & \sqsubseteq & v \\ \texttt{where} \ \theta(\mathit{true}) & \coloneqq & \{0\} \\ \theta(\mathit{false}) & \coloneqq & \{i \mid i \in \mathbb{Z} \, ; \, i \neq 0\} \end{array}$$

Closures must follow similar laws, inducing a Galois connection between $\wp(\mathit{Clo})$ and Val:

$$\{c\} \subseteq \mathit{clo\text{-}E}(\mathit{cloI}(c))$$
$$\bigsqcup_{c \in \mathit{clo\text{-}E}(v)} \mathit{clo\text{-}I}(c) \sqsubseteq v$$

Finally, $\delta[\![\_]\!]$ must be sound w.r.t. the Galois connection between concrete values and Val:

$$\mathit{int\text{-}I}(i_1 + i_2) \sqsubseteq \delta[\![+]\!](\mathit{int\text{-}I}(i_1), \mathit{int\text{-}I}(i_2))$$
$$\mathit{int\text{-}I}(i_1 - i_2) \sqsubseteq \delta[\![-]\!](\mathit{int\text{-}I}(i_1), \mathit{int\text{-}I}(i_2))$$

Supporting additional primitive types like booleans, lists, or arbitrary inductive datatypes is analogous. Introduction functions inject the type into Val and elimination functions project a finite set of discrete observations. Introduction, elimination and δ operators must all be sound and complete following a Galois connection discipline.

5.4.3 Abstract Time

The interface we use for abstract time is familiar from Van Horn and Might [2010], which introduces abstract time as a single parameter to control various forms of context sensitivity, and Smaragdakis et al. [2011], which instantiates the parameter to achieve various forms of object sensitivity. We only demonstrate call-site sensitivity in this presentation; our semantics-independent Haskell library supports object sensitivity following the same methodology.

Abstract time need only support a single operation: *tick*:

$$\mathit{Time} \ : \ \mathit{Type} \qquad\qquad \mathit{tick} \ : \ \mathit{Exp} \times \mathit{KAddr} \times \mathit{Time} \to \mathit{Time}$$

Remarkably, we need not state laws for *tick*. The interpreter will merge values which reside at the same address to preserve soundness. Therefore, any supplied implementations of *tick* is valid from a soundness perspective. However, different choices in *tick* will yield different trade-offs in precision and performance of the abstract interpreter.

5.5 The Interpreter

We now present a monadic interpreter for $\lambda\texttt{IF}$ parameterized over m, *Val* and *Time* from Section 5.4. We instantiate these parameters to obtain an analysis in Section 5.6.

We translate $A[\![\_]\!]$, a partial denotation function, to $A^m[\![\_]\!]$, a total monadic denotation function, shown in Figure 5.4.

Next we implement step^m, a *monadic* small-step *function* for compound expressions, also shown in Figure 5.4. step^m is a translation of $\_ \rightsquigarrow \_$ from a relation to a monadic function with state and nondeterminism effects.

step^m uses *push* and *pop* for manipulating stack frames, $\uparrow_p$ for lifting values from $\wp$ into m, *refine* for value refinement after branching, and a monadic version of *tick* called tick^m, each shown in Figure 5.5. Frames are pushed when the control expression e is compound and popped when e is atomic. The interpreter looks deterministic, however the nondeterminism is hidden behind $\uparrow_p$ and monadic bind

$$
\begin{array}{l}
A^m[\![\_]\!] \;:\; \mathit{Atom} \to m(\mathit{Val}) \\
A^m[\![i]\!] \;\coloneqq\; \mathit{return}(\mathit{int\text{-}I}(i)) \\
A^m[\![x]\!] \;\coloneqq\; \texttt{do} \\
\quad \rho \leftarrow \mathit{get\text{-}Env}\,;\; \sigma \leftarrow \mathit{get\text{-}Store} \\
\quad \texttt{if}\ x \in \rho\ \texttt{then}\ \mathit{return}(\sigma(\rho(x)))\ \texttt{else}\ \mathit{return}(\bot) \\
A^m[\![\lambda x.e]\!] \;\coloneqq\; \rho \leftarrow \mathit{get\text{-}Env}\,;\; \mathit{return}(\mathit{clo\text{-}I}(\langle \lambda x.e, \rho \rangle)) \\
\\
\mathit{step}^m \;:\; \mathit{Exp} \to m(\mathit{Exp}) \\
\mathit{step}^m(e) \;\coloneqq\; \texttt{do} \\
\quad \mathit{tick}^m(e)\,;\; \rho \leftarrow \mathit{get\text{-}Env} \\
\quad e' \leftarrow \texttt{case}\ e\ \texttt{of} \\
\qquad e_1 \odot e_2 \to \mathit{push}(\langle \square \odot e_2, \rho \rangle)\,;\; \mathit{return}(e_1) \\
\qquad \texttt{if0}(e_1)\{e_2\}\{e_3\} \to \mathit{push}(\langle \texttt{if0}(\square)\{e_2\}\{e_3\}, \rho \rangle)\,;\; \mathit{return}(e_1) \\
\qquad a \to \texttt{do} \\
\qquad\quad v \leftarrow A^m[\![a]\!]\,;\; fr \leftarrow \mathit{pop} \\
\qquad\quad \texttt{case}\ fr\ \texttt{of} \\
\qquad\qquad \langle \square \odot e, \rho' \rangle \to \mathit{put\text{-}Env}(\rho')\,;\; \mathit{push}(\langle v \odot \square \rangle)\,;\; \mathit{return}(e) \\
\qquad\qquad \langle v' @ \square \rangle \to \texttt{do} \\
\qquad\qquad\quad \tau \leftarrow \mathit{get\text{-}Time}\,;\; \sigma \leftarrow \mathit{get\text{-}Store} \\
\qquad\qquad\quad \langle \lambda x.e, \rho' \rangle \leftarrow \uparrow_p(\mathit{clo\text{-}E}(v')) \\
\qquad\qquad\quad \mathit{put\text{-}Env}(\rho'[x \mapsto \langle x, \tau \rangle])\,;\; \mathit{put\text{-}Store}(\sigma \sqcup [\langle x, \tau \rangle \mapsto v])\,;\; \mathit{return}(e) \\
\qquad\qquad \langle v' \oplus \square \rangle \to \mathit{return}(\delta[\![\oplus]\!](v', v)) \\
\qquad\qquad \langle \texttt{if0}(\square)\{e_1\}\{e_2\}, \rho' \rangle \to \texttt{do} \\
\qquad\qquad\quad \mathit{put\text{-}Env}(\rho')\,;\; b \leftarrow \uparrow_p(\mathit{if0\text{-}E}(v))\,;\; \mathit{refine}(a, b) \\
\qquad\qquad\quad \texttt{if}(b)\ \texttt{then}\ \mathit{return}(e_1)\ \texttt{else}\ \mathit{return}(e_2) \\
\quad \mathit{gc}(e')\,;\; \mathit{return}(e')
\end{array}
$$

Figure 5.4: Monadic Semantics

operations $x \leftarrow e_1 \; ; \; e_2$. The use of *refine* enforces a limited form of path-condition, and will yield each variation of path and flow sensitivity given the appropriate monad.

We implement abstract garbage collection *gc* in a general way using the monadic effect interface, also shown in Figure 5.5. R and KR are as defined in Section 5.2. Remarkably, this single implementation supports instantiation to analyses with varying path and flow sensitivities.

Preserving Soundness In the monadic interpreter, updates to both the data-store and stack-store must merge rather than overwrite values. To support $\sqcup$ for the stack store we redefine the domain to map to a powerset of frames:

$$\kappa\sigma \in \mathit{KStore} \; : \; \mathit{KAddr} \to \wp(\mathit{Frame} \times \mathit{KAddr})$$

Execution In the concrete semantics, execution takes the form of a least-fixed-point computation over the collecting semantics *collect*. This in general requires a join-semilattice structure for some Σ and a transition system $\Sigma \to \Sigma$. However, we no longer have a transition system $\Sigma \to \Sigma$; we have a monadic function $\mathit{Exp} \to m(\mathit{Exp})$ which cannot be iterated to least-fixed-point to execute the analysis.

To solve this we require the existence of a Galois connection between monadic actions and some transition system: $\Sigma \to \Sigma \underset{\alpha^{\Sigma \leftrightarrow m}}{\overset{\gamma^{\Sigma \leftrightarrow m}}{\leftrightarrows}} \mathit{Exp} \to m(\mathit{Exp})$. This Galois connection allows us to implement the analysis by transporting our interpreter to the transition system $\Sigma \to \Sigma$ through $\gamma^{\Sigma \leftrightarrow m}$, and then iterating to fixed-point in Σ. Furthermore, it serves to *transport other Galois connections* as part of our correctness framework. This will allow us to construct Galois connections between

$$
\begin{array}{l}
\mathit{push} \;:\; \mathit{Frame} \to m(\mathit{unit}) \\
\mathit{push}(fr) \;\coloneqq\; \texttt{do} \\
\quad \kappa l \leftarrow \mathit{get\text{-}KAddr}\,;\; \kappa\sigma \leftarrow \mathit{get\text{-}KStore}\,;\; \kappa l' \leftarrow \mathit{get\text{-}Time} \\
\quad \mathit{put\text{-}KStore}(\kappa\sigma \sqcup [\kappa l' \mapsto \{fr :: \kappa l\}])\;;\; \mathit{put\text{-}KAddr}(\kappa l') \\
\mathit{pop} \;:\; m(\mathit{Frame}) \\
\mathit{pop} \;\coloneqq\; \texttt{do} \\
\quad \kappa l \leftarrow \mathit{get\text{-}KAddr}\,;\; \kappa\sigma \leftarrow \mathit{get\text{-}KStore}\,;\; fr :: \kappa l' \leftarrow \uparrow_p(\kappa\sigma(\kappa l)) \\
\quad \mathit{put\text{-}KAddr}(\kappa l')\;;\; \mathit{return}(fr) \\
\uparrow_p \;:\; \forall A.\wp(A) \to m(A) \\
\uparrow_p(\{a_1, \ldots, a_n\}) \;\coloneqq\; \mathit{return}(a_1) \boxplus \cdots \boxplus \mathit{return}(a_n) \\
\mathit{refine} \;:\; \mathit{Atom} \times \mathit{Bool} \to m(\mathit{unit}) \\
\mathit{refine}(i, b) \;\coloneqq\; \mathit{return}(\mathit{unit}) \\
\mathit{refine}(x, b) \;\coloneqq\; \texttt{do} \\
\quad \rho \leftarrow \mathit{get\text{-}Env}\,;\; \sigma \leftarrow \mathit{get\text{-}Store} \\
\quad \mathit{put\text{-}Store}(\sigma[\rho(x) \mapsto b]) \\
\mathit{tick}^m \;:\; \mathit{Exp} \to m(\mathit{unit}) \\
\mathit{tick}^m(e) \;\coloneqq\; \texttt{do} \\
\quad \tau \leftarrow \mathit{get\text{-}Time}\,;\; \kappa l \leftarrow \mathit{get\text{-}KAddr} \\
\quad \mathit{put\text{-}Time}(\mathit{tick}(e, \kappa l, \tau)) \\
\mathit{gc} \;:\; \mathit{Exp} \to m(\mathit{unit}) \\
\mathit{gc}(e) \;\coloneqq\; \texttt{do} \\
\quad \rho \leftarrow \mathit{get\text{-}Env}\,;\; \sigma \leftarrow \mathit{get\text{-}Store} \\
\quad \kappa l \leftarrow \mathit{get\text{-}KAddr}\,;\; \kappa\sigma \leftarrow \mathit{get\text{-}KStore} \\
\quad \mathit{put\text{-}KStore}(\{\kappa l \mapsto \kappa\sigma(\kappa l) \mid \kappa l \in \mathit{KR}(\kappa l, \kappa\sigma)\}) \\
\quad \mathit{put\text{-}Store}(\{l \mapsto \sigma(l) \mid l \in R(e, \rho, \sigma, \kappa l, \kappa\sigma)\})
\end{array}
$$

Figure 5.5: Monadic helper functions

monads $m_1 \underset{\alpha^m}{\overset{\gamma^m}{\leftrightarrows}} m_2$ and derive Galois connections between transition systems $\Sigma_1 \underset{\alpha^\Sigma}{\overset{\gamma^\Sigma}{\leftrightarrows}} \Sigma_2$.

An execution of our interpreter is then the least-fixed-point iteration of $step^m$ transported through $\gamma^{\Sigma \leftrightarrow m}$:

$$analysis \coloneqq \mu X.X \sqcup \varsigma_0 \sqcup \gamma^{\Sigma \leftrightarrow m}(step^m)(X)$$

where ς_0 is the injection of the initial program e_0 into Σ and $\gamma^{\Sigma \leftrightarrow m}$ has type $(Exp \to m(Exp)) \to \Sigma \to \Sigma$.

5.6 Recovering Analyses

In Section 5.5, we defined a monadic interpreter with the uninstantiated parameters from Section 5.4: m, Val and $Time$. To recover a concrete interpreter, we instantiate these parameters to concrete components $M^\natural$, $Val^\natural$ and $Time^\natural$, and to recover an abstract interpreter we instantiate them to abstract components $M^\sharp$, $Val^\sharp$ and $Time^\sharp$. Furthermore, the concrete transition system $\Sigma^\natural$ induced by $M^\natural$ will recover the collecting semantics, which is our final target of abstraction, and the resulting analysis will take the form of an abstract transition system $\Sigma^\sharp$ induced by $M^\sharp$.

5.6.1 Recovering a Concrete Interpreter

To recover a concrete interpreter, we instantiate the generic monadic interpreter from Section 5.5 with concrete parameters $Val^\natural$, $\delta^\natural$, $Time^\natural$ and $M^\natural$, shown in figures 5.6 and 5.7.

$$v \in \quad Val^\sharp \;\coloneqq\; \wp(Clo^\sharp \cup \mathbb{Z})$$
$$\tau \in Time^\sharp \;\coloneqq\; (Exp \times KAddr^\sharp)^*$$

$$\textit{int-I}^\sharp \;:\; \mathbb{Z} \to Val^\sharp$$
$$\textit{int-I}^\sharp(i) \;\coloneqq\; \{i\}$$
$$\textit{if0-E}^\sharp \;:\; Val^\sharp \to \wp(Bool)$$
$$\textit{if0-E}^\sharp(v) \;\coloneqq\; \{\textit{true} \mid 0 \in v\} \cup \{\textit{false} \mid \exists i \in v \;;\; i \neq 0\}$$
$$\textit{Clo-I}^\sharp \;:\; Clo^\sharp \to Val^\sharp$$
$$\textit{Clo-I}^\sharp(c) \;\coloneqq\; \{c\}$$
$$\textit{Clo-E}^\sharp \;:\; Val^\sharp \to \wp(Clo^\sharp)$$
$$\textit{Clo-E}^\sharp(v) \;\coloneqq\; \{c \mid c \in v\}$$
$$\delta^\sharp \;:\; Val^\sharp \times Val^\sharp \to Val^\sharp$$
$$\delta^\sharp[\![+]\!](v_1, v_2) \;\coloneqq\; \{i_1 + i_2 \mid i_1 \in v_1 \;;\; i_2 \in v_2\}$$
$$\delta^\sharp[\![-]\!](v_1, v_2) \;\coloneqq\; \{i_1 - i_2 \mid i_1 \in v_1 \;;\; i_2 \in v_2\}$$
$$\textit{tick}^\sharp \;:\; Exp \times Time^\sharp \to Time^\sharp$$
$$\textit{tick}^\sharp(e, \kappa l, \tau) \;\coloneqq\; \langle e, \kappa l \rangle :: \tau$$

Figure 5.6: Concrete Interpreter Values and Time

$$
\begin{array}{rrcl}
\psi \in & \Psi^\sharp & \coloneqq & \mathit{Env}^\sharp \times \mathit{KAddr}^\sharp \times \mathit{KStore}^\sharp \times \mathit{Time}^\sharp \\
X \in & M^\sharp(A) & \coloneqq & \Psi^\sharp \times \mathit{Store}^\sharp \to \wp(A \times \Psi^\sharp \times \mathit{Store}^\sharp) \\
\varsigma \in & \Sigma^\sharp & \coloneqq & \wp(\mathit{Exp} \times \Psi^\sharp \times \mathit{Store}^\sharp)
\end{array}
$$

$$
\begin{array}{l}
\mathit{return}^\sharp \ : \ \forall A.A \to M^\sharp(A) \\
\mathit{return}^\sharp(x)(\psi, s) \ \coloneqq \ \{\langle x, \psi, s\rangle\} \\
\mathit{bind}^\sharp \ : \ \forall AB.M^\sharp(A) \to (A \to M^\sharp(B)) \to M^\sharp(B) \\
\mathit{bind}^\sharp(X)(f)(\psi, \sigma) \ \coloneqq \ \bigcup_{\langle x, \psi', \sigma'\rangle \in X(\psi, \sigma)} f(x)(\psi', \sigma') \\
\mathit{get\text{-}env}^\sharp \ : \ M^\sharp(\mathit{Env}^\sharp) \\
\mathit{get\text{-}env}^\sharp(\langle \rho, \kappa l, \kappa\sigma, \tau\rangle, \sigma) \ \coloneqq \ \{\langle \rho, \langle \rho, \kappa l, \kappa\sigma, \tau\rangle, \sigma\rangle\} \\
\mathit{put\text{-}Env}^\sharp \ : \ \mathit{Env}^\sharp \to M^\sharp(\mathit{unit}) \\
\mathit{put\text{-}Env}^\sharp(\rho')(\langle \rho, \kappa l, \kappa\sigma, \tau\rangle, \sigma) \ \coloneqq \ \{\langle \bullet, \langle \rho', \sigma, \kappa, \tau\rangle, \sigma\rangle\} \\
\mathit{mzero}^\sharp \ : \ \forall A.M^\sharp(A) \\
\mathit{mzero}^\sharp(\psi, \sigma) \ \coloneqq \ \{\} \\
\_ \boxplus^\sharp \_ \ : \ \forall A.M^\sharp(A) \times M^\sharp(A) \to M^\sharp(A) \\
(X_1 \boxplus^\sharp X_2)(\psi, \sigma) \ \coloneqq \ X_1(\psi, \sigma) \cup X_2(\psi, \sigma) \\
\alpha^{\Sigma^\sharp \leftrightarrow M^\sharp} \ : \ (\Sigma^\sharp \to \Sigma^\sharp) \to \mathit{Exp} \to M^\sharp(\mathit{Exp}) \\
\alpha^{\Sigma^\sharp \leftrightarrow M^\sharp}(f)(e)(\psi, \sigma) \ \coloneqq \ f(\{\langle e, \psi, \sigma\rangle\}) \\
\gamma^{\Sigma^\sharp \leftrightarrow M^\sharp} \ : \ (\mathit{Exp} \to M^\sharp(\mathit{Exp})) \to \Sigma^\sharp \to \Sigma^\sharp \\
\gamma^{\Sigma^\sharp \leftrightarrow M^\sharp}(f)(e\psi\sigma^*) \ \coloneqq \ \bigcup_{\langle e, \psi, \sigma\rangle \in e\psi\sigma^*} f(e)(\psi, \sigma)
\end{array}
$$

Figure 5.7: Concrete Interpreter Monad

THE CONCRETE DOMAIN We instantiate *Val* to $Val^\natural$, a powerset of concrete values. $Val^\natural$ has precise introduction and elimination functions $int\text{-}I^\natural$, $if0\text{-}E^\natural$, $Clo\text{-}I^\natural$ and $Clo\text{-}E^\natural$, and primitive operator denotation $\delta^\natural$.

CONCRETE TIME We instantiate *Time* to $Time^\natural$, which captures the execution context as a sequence of previously visited expressions. $tick^\natural$ is then a cons operation.

THE CONCRETE MONAD We instantiate m to $M^\natural$, a powerset of concrete state space components. Monadic operators $bind^\natural$ and $return^\natural$ encapsulate both state-passing and set-flattening. State effects return singleton sets and nondeterminism effects are implemented with set union.

CONCRETE EXECUTION To execute the interpreter we establish the Galois connection $\Sigma^\natural \to \Sigma^\natural \xleftarrow[\alpha^{\Sigma^\natural \leftrightarrow M^\natural}]{\gamma^{\Sigma^\natural \leftrightarrow M^\natural}} Exp \to M^\natural(Exp)$ and transport the monadic interpreter through $\gamma^{\Sigma^\natural \leftrightarrow M^\natural}$. The injection for a program e_0 into $\Sigma^\natural$ is $\varsigma_0 \coloneqq \{\langle e_0, \bot, \bot, , \bot, \rangle\}$.

5.6.2 Recovering an Abstract Interpreter

To recover an abstract interpreter we instantiate the generic monadic interpreter from Section 5.5 with abstract parameters $Val^\sharp$, $\delta^\sharp$, $Time^\sharp$ and $M^\sharp$, shown in Figure 5.8. The abstract monad operators, effects and transition system are not shown for $M^\sharp$; they are identical to $M^\natural$ but with abstract components.

THE ABSTRACT DOMAIN We pick a simple abstraction for integers, $\{-, 0, +\}$, although our technique scales to other abstract domains. Abstract values $Val^\sharp$

$$
\begin{aligned}
v \in \quad & \mathit{Val}^\sharp &&\coloneqq \wp(\mathit{Clo}^\sharp \cup \{-, 0, +\}) \\
\tau \in \quad & \mathit{Time}^\sharp &&\coloneqq (\mathit{Exp} \times \mathit{KAddr}^\sharp)^{*_k} \\
\psi \in \quad & \Psi^\sharp &&\coloneqq \mathit{Env}^\sharp \times \mathit{KAddr}^\sharp \times \mathit{KStore}^\sharp \times \mathit{Time}^\sharp \\
X \in M^\sharp(A) \quad & &&\coloneqq \Psi^\sharp \times \mathit{Store}^\sharp \to \wp(A \times \Psi^\sharp \times \mathit{Store}^\sharp) \\
\varsigma \in \quad & \Sigma^\sharp &&\coloneqq \wp(\mathit{Exp} \times \Psi^\sharp \times \mathit{Store}^\sharp)
\end{aligned}
$$

$$
\begin{aligned}
\mathit{int\text{-}I}^\sharp &: \mathbb{Z} \to \mathit{Val}^\sharp \\
\mathit{if0\text{-}E}^\sharp &: \mathit{Val}^\sharp \to \wp(\mathit{Bool}) \\
\mathit{Clo\text{-}I}^\sharp &: \mathit{Clo}^\sharp \to \mathit{Val}^\sharp \\
\mathit{Clo\text{-}E}^\sharp &: \mathit{Val}^\sharp \to \wp(\mathit{Clo})
\end{aligned}
$$

$$
\begin{aligned}
\mathit{int\text{-}I}^\sharp(i) &\coloneqq \begin{cases} \{-\} & \texttt{if} \quad i < 0 \\ \{0\} & \texttt{if} \quad i = 0 \\ \{+\} & \texttt{if} \quad i > 0 \end{cases} \\
\mathit{if0\text{-}E}^\sharp(v) &\coloneqq \bigcup \begin{cases} \{\mathit{true}\} & \texttt{when} \quad 0 \in v \\ \{\mathit{false}\} & \texttt{when} \quad - \in v \vee + \in v \end{cases} \\
\mathit{Clo\text{-}I}^\sharp(c) &\coloneqq \{c\} \\
\mathit{Clo\text{-}E}^\sharp(v) &\coloneqq \{c \mid c \in v\}
\end{aligned}
$$

$$
\delta^\sharp : \mathit{Val}^\sharp \times \mathit{Val}^\sharp \to \mathit{Val}^\sharp
$$

$$
\delta^\sharp[\![+]\!](v_1, v_2) \coloneqq \bigcup \begin{cases} \{i \mid i \in v_2\} & \texttt{when} \quad 0 \in v_1 \\ \{i \mid i \in v_1\} & \texttt{when} \quad 0 \in v_2 \\ \{+\} & \texttt{when} \quad + \in v_1 \wedge + \in v_2 \\ \{-\} & \texttt{when} \quad - \in v_1 \wedge 0 \in v_2 \\ \{-, 0, +\} & \texttt{when} \quad + \in v_1 \wedge - \in v_2 \\ \{-, 0, +\} & \texttt{when} \quad - \in v_1 \wedge + \in v_2 \end{cases}
$$

$$
\delta^\sharp[\![-]\!](v_1, v_2) \coloneqq \ldots \text{analogous} \ldots
$$

$$
\begin{aligned}
\mathit{tick}^\sharp &: \mathit{Exp} \times \mathit{Time}^\sharp \to \mathit{Time}^\sharp \\
\mathit{tick}^\sharp(e, \kappa l, \tau) &\coloneqq \lfloor \langle e, \kappa l \rangle :: \tau \rfloor_k
\end{aligned}
$$

Figure 5.8: Abstract Interpreter Parameters

is defined as a powerset of abstract values. $Val^\sharp$ has introduction and elimination functions $int\text{-}I^\sharp$, $if0\text{-}E^\sharp$, $clo\text{-}I^\sharp$ and $clo\text{-}E^\sharp$, and primitive operator denotation $\delta^\sharp$. $if0\text{-}E^\sharp$ and $\delta^\sharp$ must be conservative, returning an upper bound of the precise results returned by their concrete counterparts.

Abstract Time Abstract time $Time^\sharp$ captures an approximation of the execution context as a finite sequence of previously visited expressions. $tick^\sharp$ is a cons operation followed by k-truncation, yielding a kCFA analysis [Van Horn and Might, 2010].

The Abstract Monad and Execution The abstract monad $M^\sharp$ is identical to $M^\natural$ up to the definition of $\Psi^\sharp$. The induced state space $\Sigma^\sharp$ is finite, and its least-fixed-point iteration will give a sound and computable analysis.

5.6.3 End-to-end Correctness

The end-to-end correctness of the abstract instantiation of the interpreter is factored into three steps: (1) proving the parameterized monadic interpreter correct for any instantiation of m, Val and $Time$; (2) constructing Galois connections $M^\natural \underset{\alpha^m}{\overset{\gamma^m}{\leftrightarrows}} M^\sharp$, $Val^\natural \underset{\alpha^v}{\overset{\gamma^v}{\leftrightarrows}} Val^\sharp$ and $Time^\natural \underset{\alpha^t}{\overset{\gamma^t}{\leftrightarrows}} Time^\sharp$ piecewise; and (3) transporting the combination of (1) and (2) from the monadic function space $A \to m(B)$ to its induced transition system $\Sigma \to \Sigma$. The benefit of our approach is that the first step is proved once and for all (for a particular semantics) against *any* instantiation of m, Val and $Time$ using the reasoning principles established in Section 5.4. Furthermore the second step can be proved in isolation of the first, and the construction of the

third step is fully systematic.

We do not give proofs for (1) or the abstractions for *Val* and *Time* for (2) in this chapter, although the details can be found in prior work [Cousot, 1999, Van Horn and Might, 2010]. Rather, we give definitions and proofs for the monad abstractions for (2) and their systematic mappings to transition systems for (3) through a compositional framework in Section 5.8.

The final correctness of the abstract interpreter is established as a partial order relationship between an abstraction of $\gamma^{\Sigma^\natural \leftrightarrow M^\natural}(step^m[M^\natural])$, which recovers the collecting semantics, and $\gamma^{\Sigma^\sharp \leftrightarrow M^\sharp}(step^m[M^\sharp])$, the induced abstract semantics:

Proposition 1.

$$\alpha^{\Sigma^\natural}(\gamma^{\Sigma^\natural \leftrightarrow M^\natural}(step^m[M^\natural])) \sqsubseteq \gamma^{\Sigma^\sharp \leftrightarrow M^\sharp}(step^m[M^\sharp])$$

The left-hand-side of the relationship is the induced "best specification" of the collecting semantics via Galois connection, and should be familiar from the literature on abstract interpretation [Cousot, 1999, Cousot and Cousot, 1979, Nielson et al., 1999]. This end-to-end correctness statement will be justified in a compositional setting in Section 5.8.

5.7 Varying Path and Flow Sensitivity

Sections 5.5 and 5.6 describe the construction of a path-sensitive analysis using our framework. In this section, we show an alternate definition for $M^\sharp$ which yields a flow-insensitive analysis. Section 5.8 will generalize the definitions from this section

into compositional components (monad transformers) in addition to introducing another definition for $M^\sharp$ which yields a flow-sensitive analysis.

Before going into the details of the flow-insensitive monad, we wish to build intuition regarding what one would expect from such a development. Recall the path-sensitive monad $M^\sharp$ and its state space $\Sigma^\sharp$ from Section 5.6:

$$\begin{aligned} M^\sharp(Exp) &:= \Psi^\sharp \times Store^\sharp \to \wp(Exp \times \Psi^\sharp \times Store^\sharp) \\ \Sigma^\sharp(Exp) &:= \wp(Exp \times \Psi^\sharp \times Store^\sharp) \end{aligned}$$

where $\Psi := Env^\sharp \times KAddr^\sharp \times KStore^\sharp \times Time^\sharp$. This is path-sensitive because $\Sigma^\sharp(Exp)$ can represent arbitrary *relations* between $(Exp \times \Psi)$ and $Store^\sharp$.

As discussed in Section 5.3, a flow-sensitive analysis will give a single set of facts per program point. This results in the following monad $M^{\sharp fs}$ and state space $\Sigma^{\sharp fs}$ which encode *finite maps* to $Store^\sharp$ rather than relations:

$$\begin{aligned} M^{\sharp fs}(Exp) &:= \Psi^\sharp \times Store^\sharp \to [Exp \times \Psi^\sharp \mapsto Store^\sharp] \\ \Sigma^{\sharp fs}(Exp) &:= [Exp \times \Psi^\sharp \mapsto Store^\sharp] \end{aligned}$$

Finally, a flow-insensitive analysis must contain a global set of facts for each variable, which we achieve by pulling $Store^\sharp$ out of the powerset:

$$\begin{aligned} M^{\sharp fi}(Exp) &:= \Psi^\sharp \times Store^\sharp \to \wp(Exp \times \Psi^\sharp) \times Store^\sharp \\ \Sigma^{\sharp fi}(Exp) &:= \wp(Exp \times \Psi^\sharp) \times Store^\sharp \end{aligned}$$

These three resulting structures, $\Sigma^\sharp$, $\Sigma^{\sharp fs}$ and $\Sigma^{\sharp fi}$, capture the essence of path-sensitive, flow-sensitive and flow-insensitive transition systems, and arise naturally from $M^\sharp$, $M^{\sharp fs}$ and $M^{\sharp fi}$, which each have monadic structure. We only describe $M^{\sharp fi}$ directly in this section; in Section 5.8 we describe a more compositional set of building blocks, from which $M^\sharp$, $M^{\sharp fs}$ and $M^{\sharp fi}$ are recovered.

5.7.1 Flow Insensitive Monad

We show the definitions for monad operators, state effects, nondeterminism effects, and mapping to transition system for the flow-insensitive monad $M^{\sharp fi}$ in Figure 5.9.

The $bind^{\sharp fi}$ operation performs the global store merging required to capture a flow-insensitive analysis. The unit for $bind^{\sharp fi}$ returns one nondeterminism branch and a single global store. State effects $get\text{-}Env^{\sharp fi}$ and $put\text{-}Env^{\sharp fi}$ return a single branch of nondeterminism. Nondeterminism operations union the powerset and join the store pairwise. Finally, the Galois connection relating $M^{\sharp fi}$ to the state space $\Sigma^{\sharp fi}$ also computes powerset unions and store joins pairwise.

Instantiating the generic monadic interpreter with $M^{\natural}$, $M^{\sharp}$ and $M^{\sharp fi}$ yields a concrete interpreter, path-sensitive abstract interpreter, and flow-insensitive abstract interpreter respectively, purely by changing the underlying monad. Furthermore, the proofs of abstraction between interpreters and their induced transition systems is isolated to a proof of abstraction between monads.

5.8 A Compositional Monadic Framework

In our development thus far, any modification to the interpreter requires redesigning the monad $M^{\sharp}$ and constructing new proofs relating $M^{\sharp}$ to $M^{\natural}$. We want to avoid reconstructing complicated monads for interpreters, especially as languages and analyses grow and change. Even more, we want to avoid reconstructing complicated *proofs* that such changes require. Toward this goal, we introduce a compositional framework for constructing monads which are correct-by-construction by extending

$$M^{\sharp fi}(A) \coloneqq \Psi^\sharp \times \mathit{Store}^\sharp \to \wp(A \times \Psi^\sharp) \times \mathit{Store}^\sharp$$
$$\varsigma \in \Sigma^{\sharp fi} \coloneqq \wp(\mathit{Exp} \times \Psi^\sharp) \times \mathit{Store}^\sharp$$

$\mathit{return}^{\sharp fi} : \forall A.A \to M^{\sharp fi}(A)$

$\mathit{return}^{\sharp fi}(x)(\psi,\sigma) \coloneqq (\{x,\psi\},\sigma)$

$\mathit{bind}^{\sharp fi} : \forall AB.M^{\sharp fi}(A) \to (A \to M^{\sharp fi}(B)) \to M^{\sharp fi}(B)$

$\mathit{bind}^{\sharp fi}(X)(f)(\psi,\sigma) \coloneqq$

$(\{y\psi_{11},\ldots,y\psi_{1m_1},\ldots,y\psi_{n1},\ldots,y\psi_{nm_n}\},\sigma_1 \sqcup \cdots \sqcup \sigma_n)$ `where`

$(\{\langle x_1,\psi_1\rangle,\ldots,\langle x_n,\psi_n\rangle\},\sigma') \coloneqq X(\psi,\sigma)$

$(\{y\psi_{i1},\ldots,y\psi_{im_i}\},\sigma_i) \coloneqq f(x_i)(\psi_i,\sigma')$

$\mathit{get\text{-}Env}^{\sharp fi} : M^{\sharp fi}(\mathit{Env}^\sharp)$

$\mathit{get\text{-}Env}^{\sharp fi}(\langle\rho,\kappa,\tau\rangle,\sigma) \coloneqq (\{(\rho,\langle\rho,\kappa,\tau\rangle)\},\sigma)$

$\mathit{put\text{-}Env}^{\sharp fi} : \mathit{Env}^\sharp \to M^{\sharp fi}(\mathit{unit})$

$\mathit{put\text{-}Env}^{\sharp fi}(\rho')(\langle\rho,\kappa,\tau\rangle,\sigma) \coloneqq (\{\langle\bullet,\langle\rho',\kappa,\tau\rangle\rangle\},\sigma)$

$\mathit{mzero}^{\sharp fi} : \forall A.M^{\sharp fi}(A)$

$\mathit{mzero}^{\sharp fi}(\psi,\sigma) \coloneqq \langle\{\},\bot\rangle$

$\_ \boxplus^{\sharp fi} \_ : \forall A.M^{\sharp fi}(A) \times M^{\sharp fi}(A) \to M^{\sharp fi}A$

$(X_1 \boxplus^{\sharp fi} X_2)(\psi,\sigma) \coloneqq (x\psi_1^* \cup x\psi_2^*,\sigma_1 \sqcup \sigma_2)$ `where`

$(x\psi_i^*,\sigma_i) \coloneqq X_i(\psi,\sigma)$

$\alpha^{\Sigma^\sharp \leftrightarrow M^{\sharp fi}} : (\Sigma^{\sharp fi} \to \Sigma^{\sharp fi}) \to \mathit{Exp} \to M^{\sharp fi}(\mathit{Exp})$

$\alpha^{\Sigma^\sharp \leftrightarrow M^{\sharp fi}}(f)(e)(\psi,\sigma) \coloneqq f(\{\langle e,\psi\rangle\},\sigma)$

$\gamma^{\Sigma^\sharp \leftrightarrow M^{\sharp fi}} : (\mathit{Exp} \to M^{\sharp fi}(\mathit{Exp})) \to \Sigma^{\sharp fi} \to \Sigma^{\sharp fi}$

$\gamma^{\Sigma^\sharp \leftrightarrow M^{\sharp fi}}(f)(e\psi^*,\sigma) \coloneqq$

$(\{e\psi_{11},\ldots,e\psi_{n1},\ldots,e\psi_{nm_n}\},\sigma_1 \sqcup \cdots \sqcup \sigma_n)$ `where`

$\{\langle e_1,\psi_1\rangle,\ldots,\langle e_n,\psi_n\rangle\} \coloneqq e\psi^*$

$(\{e\psi_{i1},\ldots,e\psi_{im_i}\},\sigma_i) \coloneqq f(e_i)(\psi_i,\sigma)$

Figure 5.9: Flow Insensitive Monad Parameter

the well-known structure of monad transformer to that of *Galois transformer*.

Galois transformers are monad transformers which transport Galois connections and mappings to an executable transition system. We make this definition precise and prove our Galois transformers correct in Section 5.8.4. For now we present monad transformer operations augmented with the computational part of Galois transformers: the mapping to a transition system, which we called $\alpha^{\Sigma^\sharp \leftrightarrow M^\sharp}$, $\gamma^{\Sigma^\sharp \leftrightarrow M^\sharp}$, $\alpha^{\Sigma^\sharp \leftrightarrow M^{\sharp fi}}$ and $\gamma^{\Sigma^\sharp \leftrightarrow M^{\sharp fi}}$ in sections 5.6 and 5.7.

There are two monadic effects used in our monadic interpreter: state and nondeterminism. For state, we review the state monad transformer $S^t[s]$, which is standard [Liang et al., 1995, Moggi, 1989], however we also show how $S^t[s]$ maps to a transition system and obeys Galois transformer properties. For nondeterminism we develop two new monad transformers: $\wp^t$ and $F^t[s]$. These monad transformers are fully general purpose, even outside the context of program analysis, and are novel in this work. Finally we show that $\wp^t$ and $F^t[s]$ map to transition systems and obey Galois transformer properties.

To create a monad with various state and nondeterminism effects, one need only construct some composition of these three monad transformers. Implementations and proofs for monadic sequencing, state effects, nondeterminism effects, and mappings to an executable transition system will come entirely for free. This means that for a language which has a different state space than the example in this chapter, no added effort is required to construct a monad stack for that language; it will merely require a different selection and permutation of the same monad transformer components.

Path and flow sensitivity properties arise from the *order of composition* of

state and nondeterminism monad transformers. Placing state after nondeterminism ($S^t[s] \circ \wp^t$ or $S^t[s] \circ F^t[s']$) will result in s being path-sensitive. Placing state before nondeterminism ($\wp^t \circ S^t[s]$ or $F^t[s'] \circ S^t[s]$) will result in s being flow-insensitive. Finally, when $F^t[s]$ is used in place of $S^t[s] \circ \wp^t$ or $\wp^t \circ S^t[s]$, s will be flow-sensitive. The combination of all three sensitivities is $M \coloneqq S^t[s_1] \circ F^t[s_2] \circ S^t[s_3]$ which induces the transition system $\Sigma(\mathit{Exp}) \coloneqq [\mathit{Exp} \times s_1 \mapsto s_2] \times s_3$, where s_1 is path-sensitive, s_2 is flow-sensitive, and s_3 is flow-insensitive. Using $S^t[s]$, $\wp^t$ and $F^t[s]$, one can easily choose which components of the state space should be path-sensitive, flow-sensitive or flow-insensitive, purely by the order of monad composition.

In the following definitions we must refer to *bind*, *return* and other operations from the underlying monad, which we notate bind^m, return^m, $\leftarrow^m$, etc.

5.8.1 State Galois Transformer

The state Galois transformer is shown in Figure 5.10. return^{S^t}, bind^{S^t}, get^{S^t} and put^{S^t} require that m be a monad. mzero^{S^t} and $\_ \boxplus^{S^t} \_$ require that m be a monad with nondeterminism effects. And finally, α^{S^t} and γ^{S^t} require that m maps to Σ^m via Galois connection $\Sigma(A) \to \Sigma(B) \xleftarrow[\alpha^m]{\gamma^m} A \to m(B)$.

5.8.2 Nondeterminism Galois Transformer

The nondeterminism Galois transformer is shown in Figure 5.11. Crucially, $\mathit{return}^{\wp^t}$ and $\mathit{bind}^{\wp^t}$ require that m be both a monad and a *join-semilattice functor*. We attribute this requirement (and the difficulty of expressing it in Haskell) as a possible

$$
\begin{aligned}
S^t[s] &: (\mathit{Type} \to \mathit{Type}) \to \mathit{Type} \to \mathit{Type} \\
S^t[s](m)(A) &\coloneqq s \to m(A \times s) \\
\Pi^{S^t}[s] &: (\mathit{Type} \to \mathit{Type}) \to \mathit{Type} \to \mathit{Type} \\
\Pi^{S^t}[s](\Sigma)(A) &\coloneqq \Sigma(A \times s)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{return}^{S^t} : \forall A.A \to S^t[s](m)(A) \\
&\mathit{return}^{S^t}(x)(s) \coloneqq \mathit{return}^m(x, s) \\
&\mathit{bind}^{S^t} : \forall AB.S^t[s](m)(A) \to (A \to S^t[s](m)(B)) \to S^t[s](m)(B) \\
&\mathit{bind}^{S^t}(X)(f)(s) \coloneqq \langle x, s' \rangle \leftarrow^m X(s)\,;\ f(x)(s') \\
&\mathit{get}^{S^t} : S^t[s](m)(s) \\
&\mathit{get}^{S^t}(s) \coloneqq \mathit{return}^m(s, s) \\
&\mathit{put}^{S^t} : s \to S^t[s](m)(\mathit{unit}) \\
&\mathit{put}^{S^t}(s')(s) \coloneqq \mathit{return}^m(\bullet, s') \\
&\mathit{mzero}^{S^t} : \forall A.S^t[s](m)(A) \\
&\mathit{mzero}^{S^t}(s) \coloneqq \mathit{mzero}^m \\
&\_ \boxplus^{S^t} \_ : \forall A.S^t[s](m)(A) \times S^t[s](m)(A) \to S^t[s](m)(A) \\
&(X_1 \boxplus^{S^t} X_2)(s) \coloneqq X_1(s) \boxplus^m X_2(s) \\
&\alpha^{S^t} : \forall AB.(\Pi^{S^t}[s](\Sigma^m)(A) \to \Pi^{S^t}[s](\Sigma^m)(B)) \to A \to S^t[s](m)(B) \\
&\alpha^{S^t}(f)(x)(s) \coloneqq \alpha^m(f)(x, s) \\
&\gamma^{S^t} : \forall AB.(A \to S^t[s](m)(B)) \to \Pi^{S^t}[s](\Sigma^m)(A) \to \Pi^{S^t}[s](\Sigma^m)(B) \\
&\gamma^{S^t}(f) \coloneqq \gamma^m(\lambda\langle x, s \rangle.f(x)(s))
\end{aligned}
$$

Figure 5.10: State Galois Transformer

reason why it has not been discovered thus far. This functorality of m is instantiated with $\wp(\_)$ using the usual join-semilattice on powersets: $\{\}$ for $\bot$ and $\cup$ for $\sqcup$. $get^{\wp^t}$ and $put^{\wp^t}$ require that m be a monad with state effects. Like the state Galois transformer, $\alpha^{\wp^t}$ and $\gamma^{\wp^t}$ require that m maps to Σ^m via Galois connection.

Lemma 3. *[$\wp^t$ laws] $bind^{\wp^t}$ and $return^{\wp^t}$ satisfy monad laws, $get^{\wp^t}$ and $put^{\wp^t}$ satisfy state monad laws, and $mzero^{\wp^t}$ and $\boxplus^{\wp^t}$ satisfy nondeterminism monad laws.*

See our proofs in Section A, where the key lemma in proving monad laws is the join-semilattice functorality of m, namely that:

$$return^m(x \sqcup y) = return^m(x) \sqcup^m return^m(y)$$
$$bind^m(X \sqcup Y)(f) = bind^m(X)(f) \sqcup^m bind^m(Y)(f)$$

5.8.3 Flow Sensitivity Galois Transformer

The flow sensitivity monad transformer, shown in Figure 5.12, is a unique monad transformer that combines state and nondeterminism effects, and does not arise naturally from composing vanilla nondeterminism and state transformers. The finite map in the definition of $F^t[s]$ is what yields flow sensitivity when instantiated to a monadic interpreter. After instantiation, $F^t[s](m)(A)$ will be $Store^\sharp \to [Exp \times \Psi^\sharp \to Store^\sharp]$, which maps each possible expression and context to a unique abstract store.

Like nondeterminism, $return^{F^t}$ and $bind^{F^t}$ require that m be both a monad and a *join-semilattice functor*. This functorality of m is instantiated with $[\_ \mapsto s]$ using the usual join-semilattice on finite maps: $\{\}$ for $\bot$ and:

$$Y \sqcup Z \coloneqq \{x \mapsto y \sqcup z \mid \{x \mapsto y\} \in X \wedge \{x \mapsto z\} \in Y\}$$

$$
\begin{aligned}
\wp^t &: (\mathit{Type} \to \mathit{Type}) \to \mathit{Type} \to \mathit{Type} \\
\wp^t(m)(A) &\coloneqq m(\wp(A)) \\
\Pi^{\wp^t} &: (\mathit{Type} \to \mathit{Type}) \to \mathit{Type} \to \mathit{Type} \\
\Pi^{\wp^t}(\Sigma)(A) &\coloneqq \Sigma(\wp(A))
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{return}^{\wp^t} : \forall A.A \to \wp^t(m)(A) \\
&\mathit{return}^{\wp^t}(x) \coloneqq \mathit{return}^m(\{x\}) \\
&\mathit{bind}^{\wp^t} : \forall AB.\wp^t(m)(A) \to (A \to \wp^t(m)(B)) \to \wp^t(m)(B) \\
&\mathit{bind}^{\wp^t}(X)(f) \coloneqq \ \texttt{do} \\
&\quad \{x_1, \ldots, x_n\} \leftarrow^m X \\
&\quad f(x_1) \sqcup^m \cdots \sqcup^m f(x_n) \\
&\mathit{get}^{\wp^t} : \wp^t(m)(s) \\
&\mathit{get}^{\wp^t} \coloneqq s \leftarrow^m \mathit{get}^m \,;\, \mathit{return}^m(\{s\}) \\
&\mathit{put}^{\wp^t} : s \to \wp^t(m)(\mathit{unit}) \\
&\mathit{put}^{\wp^t}(s) \coloneqq u \leftarrow^m \mathit{put}^m(x) \,;\, \mathit{return}^m(\{u\}) \\
&\mathit{mzero}^{\wp^t} : \forall A.\wp^t(m)(A) \\
&\mathit{mzero}^{\wp^t} \coloneqq \bot^m \\
&\_ \boxplus^{\wp^t} \_ : \forall A.\wp^t(m)(A) x \wp^t(m)(A) \to \wp^t(m)(A) \\
&X_1 \boxplus^{\wp^t} X_2 \coloneqq X_1 \sqcup^m X_2 \\
&\alpha^{\wp^t} : \forall AB.(\Pi^{\wp^t}(\Sigma^m)(A) \to \Pi^{\wp^t}(\Sigma^m)(B)) \to A \to \wp^t(m)(B) \\
&\alpha^{\wp^t}(f)(x) \coloneqq \alpha^m(f)(\{x\}) \\
&\gamma^{\wp^t} : \forall AB.(A \to \wp^t(m)(B)) \to \Pi^{\wp^t}(\Sigma^m)(A) \to \Pi^{\wp^t}(\Sigma^m)(B) \\
&\gamma^{\wp^t}(f) \coloneqq \gamma^m(\lambda\{x_1, \ldots, x_n\}.f(x_1) \sqcup^m \cdots \sqcup^m f(x_n))
\end{aligned}
$$

Figure 5.11: Nondeterminism Galois Transformer

$$
\begin{aligned}
F^t[s] &: (\mathit{Type} \to \mathit{Type}) \to \mathit{Type} \to \mathit{Type} \\
F^t[s](m)(A) &\coloneqq s \to m([A \mapsto s]) \\
\Pi^{F^t}[s] &: (\mathit{Type} \to \mathit{Type}) \to \mathit{Type} \to \mathit{Type} \\
\Pi^{F^t}[s](\Sigma)(A) &\coloneqq \Sigma([A \mapsto s])
\end{aligned}
$$

$\mathit{return}^{F^t} : \forall A.A \to F^t[s](m)(A)$

$\mathit{return}^{F^t}(x)(s) \coloneqq \mathit{return}^m(\{x \mapsto s\})$

$\mathit{bind}^{F^t} : \forall AB.F^t[s](m)(A) \to (A \to F^t[s](m)(B)) \to F^t[s](m)(B)$

$\mathit{bind}^{F^t}(X)(f)(s) \coloneqq$ `do`

$\{x_1 \mapsto s_1, \ldots, x_n \mapsto s_n\} \leftarrow^m X(s)$

$f(x_1)(s_1) \sqcup^m \cdots \sqcup^m f(x_n)(s_n)$

$\mathit{get}^{F^t} : F^t[s](m)(s)$

$\mathit{get}^{F^t}(s) \coloneqq \mathit{return}^m(\{s \mapsto s\})$

$\mathit{put}^{F^t} : s \to F^t[s](m)(\mathit{unit})$

$\mathit{put}^{F^t}(s')(s) \coloneqq \mathit{return}^m(\{\bullet \mapsto s'\})$

$\mathit{mzero}^{F^t} : \forall A.F^t[s](m)(A)$

$\mathit{mzero}^{F^t}(s) \coloneqq \bot^m$

$\_ \boxplus^{F^t} \_ : \forall A.F^t[s](m)(A) x F^t[s](m)(A) \to F^t[s](m)(A)$

$(X_1 \boxplus^{F^t} X_2)(s) \coloneqq X_1(s) \sqcup^m X_2(s)$

$\alpha^{F^t} : \forall AB.(\Pi^{F^t}[s](\Sigma^m)(A) \to \Pi^{F^t}[s](\Sigma^m)(B)) \to A \to F^t[s](m)(B)$

$\alpha^{F^t}(f)(x)(s) \coloneqq \alpha^m(f)(\{x \mapsto s\})$

$\gamma^{F^t} : \forall AB.(A \to F^t[s](m)(B)) \to \Pi^{F^t}[s](\Sigma^m)(A) \to \Pi^{F^t}[s](\Sigma^m)(B)$

$\gamma^{F^t}(f) \coloneqq \gamma^m(\lambda\{x_1 \mapsto s_1, \ldots, x_n \mapsto s_n\}.f(x_1)(s_1) \sqcup^m \cdots \sqcup^m f(x_n)(s_n))$

Figure 5.12: Flow Sensitivity Galois Transformer

$get^{\wp^t}$ and $put^{\wp^t}$ require that m be a monad. Like the nondeterminism Galois transformer, $\alpha^{\wp^t}$ and $\gamma^{\wp^t}$ require that m maps to Σ^m via Galois connection.

Lemma 4. *[F^t laws] $bind^{F^t}$ and $return^{F^t}$ satisfy monad laws, get^{F^t} and put^{F^t} satisfy state monad laws, and $mzero^{F^t}$ and $\boxplus^{F^t}$ satisfy nondeterminism monad laws.*

See our proofs in A. Monad and nondeterminism laws are are analogous to those for nondeterminism, and also rely on the join-semilattice functorailty of m. State monad laws are proved by calculation.

5.8.4 Galois Transformers

The capstone of our framework is the fact that monad transformers $S^t[s]$, $\wp^t$ and $F^t[s]$ are also *Galois transformers*.

Definition 1. *A monad transformer T is a Galois transformer with transition system Π if:*

1. *T transports a Galois connection between monads m_1 and m_2 into a Galois connection between $T(m_1)$ and $T(m_2)$:*

$$\begin{array}{ccc} A \to m_2(B) & \xrightarrow[T[m_2]]{} & A \to T(m_2)(B) \\ \alpha^m \uparrow \downarrow \gamma^m & & T[\alpha^m] \uparrow \downarrow T[\gamma^m] \\ A \to m_1(B) & \xrightarrow[T[m_1]]{} & A \to T(m_1)(B) \end{array}$$

   $T[m]$ must be monotonic, and T must commute with Galois connections, that is for all $f : A \to m_1(B)$:

$$T[m_2](\alpha^m(f)) = T[\alpha^m](T[m_1](f))$$

2. Π *transports Galois connections between induced transition systems* Σ_1 *and* Σ_2 *into Galois connections between* $\Pi(\Sigma_1)$ *and* $\Pi(\Sigma_2)$*:*

$$\begin{array}{ccc} \Sigma_2(A) \to \Sigma_2(B) & \xrightarrow[\Pi[\Sigma_2]]{} & \Pi(\Sigma_2)(A) \to \Pi(\Sigma_2)(B) \\ \alpha^{\Sigma} \uparrow \quad \downarrow \gamma^{\Sigma} & & \Pi[\alpha^{\Sigma}] \uparrow \quad \downarrow \Pi[\gamma^{\Sigma}] \\ \Sigma_1(A) \to \Sigma_1(B) & \xrightarrow[\Pi[\Sigma_1]]{} & \Pi(\Sigma_1)(A) \to \Pi(\Sigma_1)(B) \end{array}$$

$\Pi[\Sigma]$ *must be monotonic, and* Π *must commute with Galois connections, that is for all* $f : \Sigma_1(A) \to \Sigma_1(B)$*:*

$$\Pi[\Sigma_2](\alpha^{\Sigma}(f)) = \Pi[\alpha^{\Sigma}](\Pi[\Sigma_1](f))$$

3. T *and* Π *transport transition system mappings between* m *and* Σ *into transition system mappings between* $T(m)$ *and* $\Pi(\Sigma)$*:*

$$\begin{array}{ccc} A \to m(B) & \xrightarrow[T[m]]{} & A \to T(m)(B) \\ \alpha^{\Sigma \leftrightarrow m} \uparrow \quad \downarrow \gamma^{\Sigma \leftrightarrow m} & & T[\alpha^{\Sigma \leftrightarrow m}] \uparrow \quad \downarrow T[\gamma^{\Sigma \leftrightarrow m}] \\ \Sigma(A) \to \Sigma(B) & \xrightarrow[\Pi[\Sigma]]{} & \Pi(\Sigma)(A) \to \Pi(\Sigma)(B) \end{array}$$

$T[\gamma^{\Sigma \leftrightarrow m}]$ *must commute asymmetrically (in the partial order) with* T *and* Π*, that is for all functions* $f : A \to m(B)$*:*

$$\Pi[\Sigma](\gamma^{\Sigma \leftrightarrow m}(f)) \sqsubseteq T[\gamma^{\Sigma \leftrightarrow m}](T[m](f))$$

Lemma 5 (Galois Transformer Properties)**.** $S^t[s]$*,* $\wp^t$ *and* $F^t[s]$ *are Galois transformers.*

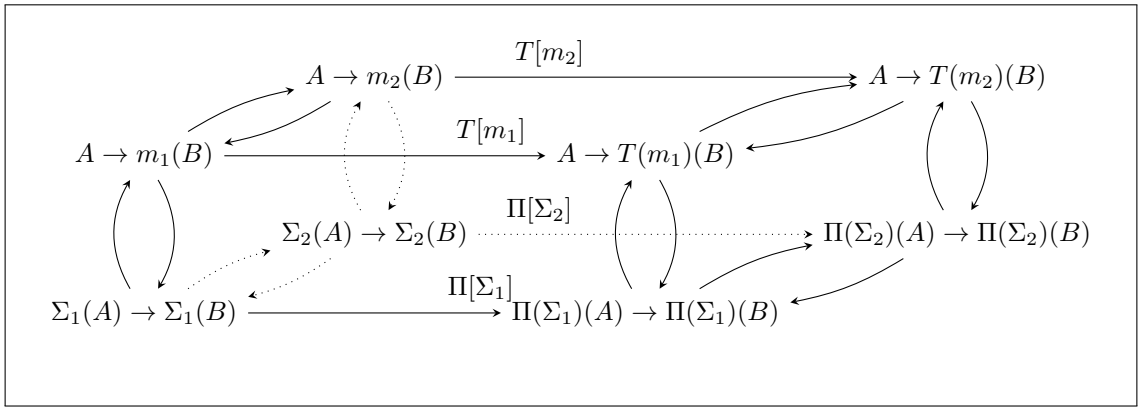


Figure 5.13: Galois Transformer Commuting Cube of Abstractions

Definitions for $\alpha^{\Sigma \leftrightarrow \gamma}$ and $\gamma^{\Sigma \leftrightarrow \gamma}$ from property (3) are shown in figures 5.10, 5.11 and 5.12. Definitions of other Galois connections and commutativity proofs are given in the appendix.

These three properties of Galois transformers snap together to form a three-dimensional diagram, shown in Figure 5.13 which relates abstractions between monads m_1 and m_2 and their transition systems Σ_1 and Σ_2 to their actions under T and Π. The left-hand side of the cube is a commuting square of abstractions between m_1, m_2, Σ_1 and Σ_2. The right-hand side of the cube is constructed from the composition of properties (1) through (3) as the front, top, back, and bottom faces of the cube, and is a commuting square of abstractions between $T(m_1)$, $T(m_2)$, $\Pi(\Sigma_1)$ and $\Pi(\Sigma_2)$. The whole cube commutes, by combining the commuting properties of the left face and the commuting properties of (1) through (3).

Theorem 5. *If T is a Galois transformer with transition system Π, given a commuting square of abstractions between monads m_1 and m_2 and their transition systems Σ_1 and Σ_2, T and Π construct a commuting square of abstractions between monads*

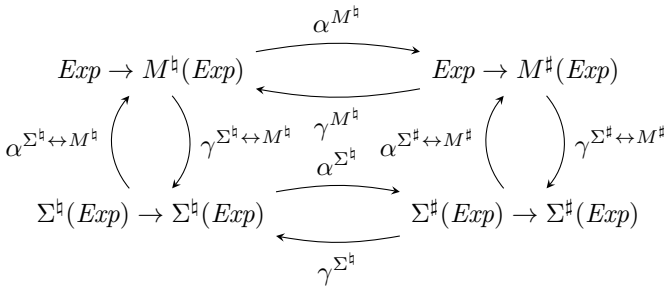
$T(m_1)$ and $T(m_2)$ and their transition systems $\Pi(\Sigma_1)$ and $\Pi(\Sigma_2)$.

The proof is the composition of Galois transformer properties, as shown in the Figure 5.13.

The consequence of this theorem is that any two compositions of Galois transformers $T_1 \circ \cdots \circ T_n$ and $U_1 \circ \cdots \circ U_n$ where U_i is an abstraction of T_i will yield a commuting square of abstractions between monads $(T_1 \circ \cdots \circ T_n)(ID)$ and $(U_1 \circ \cdots \circ U_n)(ID)$ and their induced transition systems $(\Pi^{T_1} \circ \cdots \circ \Pi^{T_n})(ID)$ and $(\Pi^{U_1} \circ \cdots \circ \Pi^{U_n})(ID)$. This is the first step in proving the resulting abstract interpreter correct; we need to establish a commuting square of abstractions between a concrete monad, an abstract monad, and their induced concrete and abstract transition systems.

5.8.5 End-to-End Correctness with Galois Transformers

In the setting of abstract interpretation, we instantiate the Galois transformer framework described above with two compositions of monad transformers yielding a commuting square of abstractions between the concrete monad $M^\natural$, the abstract monad $M^\sharp$, and concrete and abstract transition systems $\Sigma^\natural$ and $\Sigma^\sharp$:



This diagram shows how to relate monadic interpreters to transition systems (the

vertical axis of the diagram), and concrete semantics to abstract semantics (the horizontal axis of the diagram). The top half is where we write the monadic interpreter, and the bottom half is where we execute the analysis as the least-fixed point of a transition system.

We use this commuting square to systematically relate a recovered collecting semantics with the induced abstract transition system in the following theorem:

Theorem 6. *Given a commuting square of abstraction between $M^\natural$, $M^\sharp$, $\Sigma^\natural$ and $\Sigma^\sharp$, and a generic monadic interpreter $step^m$, if $collect = \gamma^{\Sigma^\natural \leftrightarrow M^\natural}(step^m[M^\natural])$ recovers the collecting semantics, then $analysis = \gamma^{\Sigma^\sharp \leftrightarrow M^\sharp}(step^m[M^\sharp])$ is a sound abstraction of the collecting semantics.*

Proof. Given that $step^m$ is monotonic in the monad parameter m, instantiating it with $M^\natural$ and $M^\sharp$ will result in:

$$\alpha^{M^\natural}(step^m[M^\natural]) \sqsubseteq step^m[M^\sharp]$$

Transporting through $\gamma^{\Sigma^\sharp \leftrightarrow M^\sharp}$, which is monotonic by virtue of forming a Galois connection with $\alpha^{\Sigma^\sharp \leftrightarrow M^\sharp}$, we have:

$$\textit{(1)}\ \gamma^{\Sigma^\sharp \leftrightarrow M^\sharp}(\alpha^{M^\natural}(step^m[M^\natural])) \sqsubseteq \gamma^{\Sigma^\sharp \leftrightarrow M^\sharp}(step^m[M^\sharp]) = analysis$$

Next, we abstract the recovered collecting semantics to form its best specification for abstraction:

$$\textit{(2)}\ \alpha^{\Sigma^\sharp}(collect) = \alpha^{\Sigma^\sharp}(\gamma^{\Sigma^\natural \leftrightarrow M^\natural}(step^m[M^\natural]))$$

Finally, we exploit the commutativity of the square of abstractions between $M^\natural$, $M^\sharp$,

$\Sigma^\natural$ and $\Sigma^\sharp$ to relate the recovered collecting semantics with the abstract monadic semantics:

$$(3)\ \ \alpha^{\Sigma^\sharp}(\gamma^{\Sigma^\natural \leftrightarrow M^\natural}(\mathit{step}^m[M^\natural])) \sqsubseteq \gamma^{\Sigma^\sharp \leftrightarrow M^\sharp}(\alpha^{M^\natural}(\mathit{step}^m[M^\natural]))$$

The transitive combination of (1), (2) and (3) establishes the soundness of the derived abstract execution system w.r.t. the recovered collecting semantics: $\alpha^{\Sigma^\sharp}(\mathit{collect}) \sqsubseteq \mathit{analysis}$. □

This theorem proves Proposition 1 in Section 5.6.3 after instantiating the example to the Galois transformer framework.

5.8.6 Applying the Framework to Our Semantics

Our setting is the ground-truth semantics $\_ \rightsquigarrow^{gc} \_$ from Section 5.2 and the generic interpreter step^m from Section 5.5.

To recover the concrete collecting semantics, we instantiate step^m to the concrete parameters for the domain and time from Section 5.6.1, and synthesize the monad as a combination of state and nondeterminism Galois transformers:

$$M^\natural \ := \ (S^t[\Psi^\natural] \circ S^t[\mathit{Store}^\natural] \circ \wp^t)(\mathit{ID})$$

To recover a path-sensitive abstract interpreter we instantiate step^m to the abstract parameters for the domain and time from Section 5.6.2, and synthesize the monad as a combination of state and nondeterminism Galois transformers:

$$M^\sharp \ := \ (S^t[\Psi^\sharp] \circ S^t[\mathit{Store}^\sharp] \circ \wp^t)(\mathit{ID})$$

which abstract $M^\natural$ piecewise. Both the implementation and correctness of the induced abstract transition system are constructed for free by theorems 5 and 6.

To recover a flow-sensitive abstract interpreter we synthesize the monad as a combination of state and flow-sensitive Galois transformers:

$$M^{\sharp fs} \coloneqq (S^t[\Psi^\sharp] \circ F^t[Store^\sharp])(ID)$$

which abstracts $M^\sharp$ piecewise.

Finally, to recover a flow-insensitive abstract interpreter we synthesize the monad as a permuted combination of state and nondeterminism Galois transformers:

$$M^{\sharp ps} \coloneqq (S^t[\Psi^\sharp] \circ \wp^t \circ S^t[Store^\sharp])(ID)$$

which abstracts $M^{\sharp ps}$ piecewise.

5.8.7 Applying the Framework to Another Semantics

Our Galois transformers framework is semantics independent, and the proofs in Section 5.8.4 need not be reproved for another semantic setting. To use our framework and establish an end-to-end correctness theorem, the user must:

- Design a generic monadic interpreter for their semantics using an interface of monadic effects
- Prove their interpreter monotonic w.r.t. parameters
- Prove that the induced concrete transition system recovers the concrete collecting semantics of interest.

The user then enjoys the following for free:

- A combination of state, nondeterminism and flow-sensitive Galois transformers which supports the monadic effect interface unique to the semantics.
- The ability to rearrange monad transformers to recover variations in path and flow sensitivities.
- An induced, executable abstract interpreter for each stack of monad transformers.
- A proof that each induced abstract interpreter is a sound abstraction of the collecting semantics, as a result of theorems 5 and 6.

5.9 Implementation

We have implemented our framework in Haskell and applied it to compute analyses for λ`IF`. Our implementation provides path sensitivity, flow sensitivity, and flow insensitivity as a semantics-independent monad library. The code shares a striking resemblance with the math.

Our implementation is suitable for prototyping and exploring the design space of static analyzers. Our analyzer supports exponentially more compositions of analysis features than any current analyzer. For example, our implementation is the first which can combine arbitrary choices in call-site, object, path and flow sensitivities. Furthermore, the user can choose different path and flow sensitivities independently for each component of the state space.

Our implementation `maam` supports command-line flags for garbage collection, mCFA, call-site sensitivity, object sensitivity, and path and flow sensitivity.

```
./maam prog.lam --gc --mcfa --kcfa=1 --ocfa=2 \
    --data-store=flow-sen --stack-store=path-sen
```

Each flag is implemented independently of each other applied to a single parameterized monadic interpreter. Furthermore, using Galois transformers allows us to prove each combination correct in one fell swoop.

A developer wishing to use our library to develop analyzers for their language of choice inherits as much of the analysis infrastructure as possible. We provide call-site, object, path and flow sensitivities as language-independent libraries. To support analysis for a new language a developer need only implement:

- A monadic semantics for their language, using state and nondeterminism effects.
- The abstract value domain, and optionally the concrete value domain if they wish to recover concrete execution.
- Intentional optimizations for their semantics like garbage collection and mcfa.

The developer then receives the following for free through our analysis library:

- A family of monads which implement their effect interface and give different path and flow sensitivities.
- Mechanisms for call-site and object sensitivities.
- An execution engine for each monad to drive the analysis.

Not only is a developer able to reuse our implementation of call-site, object, path and flow sensitivities, they need not understand the execution machinery or soundness proofs for them either. They need only verify that their monadic semantics is monotonic w.r.t. the analysis parameters, and that their abstract value domain forms a Galois connection. The execution and correctness of the final analyzer is constructed automatically given these two properties.

Our implementation is publicly available and can be installed as a cabal package: `cabal install maam`.

5.10 Related Work

Overview Program analysis comes in many forms such as points-to [Andersen, 1994], flow [Jones, 1981], or shape analysis [Chase et al., 1990], and the literature is vast. (See Hind [2001], Midtgaard [2012] for surveys.) Much of the research has focused on developing families or frameworks of analyses that endow the abstraction with a number of knobs, levers, and dials to tune precision and compute efficiently (some examples include Milanova et al. [2005], Nielson and Nielson [1997], Shivers [1991], Van Horn and Might [2010]; there are many more). These parameters come in various forms with overloaded meanings such as object [Milanova et al., 2005, Smaragdakis et al., 2011], context [Sharir and Pnueli, 1981, Shivers, 1991], path [Das et al., 2002], and heap [Van Horn and Might, 2010] sensitivities, or some combination thereof [Kastrinis and Smaragdakis, 2013].

These various forms can all be cast in the theory of abstraction interpretation

of Cousot and Cousot [1977, 1979] and understood as computable approximations of an underlying concrete interpreter. Our work demonstrates that if this underlying concrete interpreter is written in monadic style, monad transformers are a useful way to organize and compose these various kinds of program abstractions in a modular and language-independent way.

This work is inspired by the trifecta combination of Cousot, Cousot and Cousot, Cousot and Cousot's theory of abstract interpretation based on Galois connections [1999, 1977, 1979], Moggi's original monad transformers [1989] which were later popularized in Liang et al.'s *Monad Transformers and Modular Interpreters* [1995], and Sergey et al.'s *Monadic Abstract Interpreters* [Sergey et al., 2013].

Liang et al. [1995] first demonstrated how monad transformers could be used to define building blocks for constructing (concrete) interpreters. Their interpreter monad *InterpM* bears a strong resemblance to ours. We show this "building blocks" approach to interpreter construction also extends to *abstract* interpreter construction using Galois transformers. Moreover, we show that these monad transformers can be proved sound via a Galois connection to their concrete counterparts, ensuring the soundness of any stack built from sound blocks of Galois transformers. Soundness proofs of various forms of analysis are notoriously brittle with respect to language and analysis features. A reusable framework of Galois transformers offers a potential way forward for a modular metatheory of program analysis.

Cousot [1999] develops a "calculational approach" to analysis design whereby analyses are not designed and then verified *post facto*, but rather derived by positing

an abstraction and calculating it from the concrete interpreter using Galois connections. These calculations are done by hand. Our approach offers the ability to automate the calculation process for a limited set of abstractions for small-step state machines, where the abstractions are correct-by-construction through the composition of monad transformers.

We build directly on the work of Abstracting Abstract Machines (AAM) by Smaragdakis et al. [2011], Van Horn and Might [2010] in our parameterization of abstract time to achieve call-site and object sensitivity. We follow the AAM philosophy of instrumenting a concrete semantics *first* and performing a systematic abstraction *second*. This greatly simplifies the Galois connection arguments during systematic abstraction, at the cost of proving the correctness of the instrumented semantics.

Monadic Abstract Interpreters Sergey et al. [2013] first introduced the concept of writing abstract interpreters in monadic style in *Monadic Abstract Interpreters* (MAI), where variations in analysis are also recovered through monads.

In MAI, the framework's interface is based on *denotation functions* for every syntactic form of the language. The denotation functions in MAI are language-specific and specialized to their example language. MAI uses a single monad stack fixed to the denotation function interface: state on top of list. New analyses are achieved through multiple denotation functions into this single monad. Analyses in MAI are all fixed to be path-sensitive, and the methodology for incorporating other path or flow properties is to surgically instrument the execution of the analysis with

a custom Galois connection. Lastly, the framework provides no reasoning principles or proofs of soundness for the resulting analysis. A user of MAI must inline the definitions of each analysis and prove each implementation correct from scratch.

Our framework is instead based on state and nondeterminism *monadic effects*. This interface comes equipped with laws, allowing one to verify the correctness of a monadic interpreter independent of a particular monad. State and nondeterminism monadic effects capture arbitrary small-step relational semantics, and are language independent. Because we place the monadic interpreter behind an interface of effects with laws, we are able to introduce language-independent monads which capture flow-sensitivity and flow-insensitivity, and we show how to compose these features with other analysis design choices. The monadic effect interface also allows us to separate the monad from the abstract domain. Finally, our framework is compositional through the use of monad transformers, and constructs execution engines and end-to-end soundness proofs for free.

Widening for Control-Flow Hardekopf et al. [2014] also introduce a unifying account of control flow properties in *Widening for Control-Flow* (WCF), which accounts for path, flow and call-site sensitivities. WCF achieves this through an instrumentation of the abstract machine's state space which is allowed to track arbitrary contextual information, up to the path-history of the entire execution. WCF also develops a modular proof framework, proving the bulk of soundness proofs for each instantiation of the instrumentation at once.

Our work achieves similar goals, although isolating path and flow sensitivity is

not our primary objective. WCF is based on a language-dependent instrumentation of the semantics, whereas we achieve variations in path and flow sensitivity through language-independent monads.

Particular strengths of WCF are the wide range of choices for control-flow sensitivity which are shown to be implementable within the design, and the modular proof framework. For example, WCF is able to account for call-site sensitivity in their design; we account for call-site sensitivity through a different mechanism.

Particular strengths of our work is the understanding of path and flow sensitivity not through instrumentation but through semantics-independent control properties of the interpreter, and also a modular proof framework, although modular in a different sense from WCF. We also show how to compose different path and flow sensitivity choices for independent components of the state space, like a flow-sensitive data-store and path-sensitive stack-store, for example.

5.11 Conclusions

We have shown that *Galois transformers*, monad transformers that transport Galois connections and mappings to an executable transition system, are effective, language-independent building blocks for constructing program analyzers, and form the basis of a modular, reusable and composable metatheory for program analysis.

In the end, we hope language independent characterizations of analysis ingredients will both facilitate the systematic construction of program analyses and bridge the gap between various communities which often work in isolation.

Chapter 6: Abstracting Definitional Interpreters

6.1 Introduction

An abstract interpreter is intended to soundly and effectively compute an over-approximation to its concrete counterpart. For higher-order languages, these concrete interpreters tend to be formulated as state-machines (*e.g.*, Jagannathan and Weeks [1995], Jagannathan et al. [1998], Midtgaard and Jensen [2008, 2009], Might and Shivers [2006b], Might and Van Horn [2011], Sergey et al. [2013], Wright and Jagannathan [1998]). There are several reasons for this choice: they operate with simple transfer functions defined over similarly simple data structures, they make explicit all aspects of the state of a computation, and computing fixed-points in the set of reachable states is straightforward. The essence of the state-machine based approach was distilled by Van Horn and Might in their "abstracting abstract machines" (AAM) technique, which provides a systematic method for constructing abstract interpreters from standard abstract machines like the CEK- or Krivine-machines [Van Horn and Might, 2010]. Language designers who would like to build abstract interpreters and program analysis tools for their language can now, in principle at least, first build a state-machine interpreter and then turn the crank to construct the approximating abstract counterpart.

A natural pair of questions that arise from this past work is to wonder:

1. Can a systematic abstraction technique similar to AAM be carried out for interpreters written, *not* as state-machines, but instead as high-level definitional interpreters, *i.e.* recursive, compositional evaluators?

2. is such a perspective fruitful?

In this chapter, we seek to answer both questions in the affirmative.

For the first question, we show the AAM recipe can be applied to definitional interpreters in a straightforward adaptation of the original method. The primary technical challenge in this new setting is handling interpreter fixed-points in a way that is both sound and always terminates—a naive abstraction of fixed-points will be sound but isn't always terminating, and a naive use of caching for fixed-points will guarantee termination but is inherently unsound. We address this technical challenge with a straightforward caching fixed-point-finding algorithm which is both sound and guaranteed to terminate when abstracting arbitrary definitional interpreters.

For the second question, we claim that the abstract definitional interpreter perspective is fruitful in two regards. The first is unsurprising: high-level abstract interpreters offer the usual beneficial properties of their concrete counterparts in terms of being re-usable and extensible. In particular, we show that abstract interpreters can be structured with monad transformers to good effect. The second regard is more surprising, and we consider its observation to be the main contribution of this chapter.

Definitional interpreters, in contrast to abstract machines, can leave aspects of

computation implicit, relying on the semantics of the defin*ing*-language to define the semantics of the defin*ed*-language, an observation made by Reynolds in his landmark paper, *Definitional Interpreters for Higher-order Programming Languages* [Reynolds, 1972]. For example, Reynolds showed it is possible to write a definitional interpreter such that it defines a call-by-value language when the metalanguage is call-by-value, and defines a call-by-name language when the metalanguage is call-by-name. Inspired by Reynolds, we show that definitional *abstract* interpreters can likewise inherit properties of the metalanguage. In particular we construct an abstract definitional interpreter where there is no explicit representation of continuations or a call stack. Instead the interpreter is written in a straightforward recursive style, and the call stack is implicitly handled by the metalangauge. What emerges from this construction is a total abstract evaluation function that soundly approximates all possible concrete executions of a given program. But remarkably, since the abstract evaluator relies on the metalanguage to manage the call stack implicitly, it is easy to observe that it introduces no approximation in the matching of calls and returns, and therefore implements a "pushdown" analysis [Earl et al., 2010, Vardoulakis and Shivers, 2011], all without the need for any explicit machinery to do so.

6.1.1 Outline

In the remainder of this chapter, we present an adaptation of the AAM method to the setting of recursively-defined, compositional evaluation functions, a.k.a. definitional interpreters. We first briefly review the basic ingredients in the AAM recipe (§ 6.2)

and then define our definitional interpreter (§ 6.3). The interpreter is largely standard, but is written in a monadic and extensible style, so as to be re-usable for various forms of semantics we examine. The AAM technique applies in a basically straightforward way by store-allocating bindings and soundly finitizing the heap. But when naively run, the interpreter will not always terminate. To solve this problem we introduce a caching strategy and a simple fixed-point computation to ensure the interpreter terminates (§ 6.4). It is at this point that we observe the interpreter we have built enjoys the "pushdown" property *à la* Reynolds—it is inherited from the defining language of our interpreter and requires no explicit mechanism (§ 6.5).

Having established the main results, we then explore some variations in brief vignettes that showcase the flexibility of our definitional abstract interpreter approach. First we consider the widely used technique of so-called "store-widening," which trades precision for efficiency by modeling the abstract store globally instead of locally (§ 6.6). Thanks to our monadic formulation of the interpreter, this is achieved by a simple re-ordering of the monad transformer stack. We also explore some alternative abstractions, showing that due to the extensible construction, it's easy to experiment with alternative components for the abstract interpreter. In particular, we define an alternative interpretation of the primitive operations that remains completely precise until forced by joins in the store to introduce approximation (§ 6.7). As another variation, we explore computing a form of symbolic execution as yet another instance of our interpreter, as well as how to incorporate so-called "abstract garbage collection," a well-known technique for improving the precision of abstract interpretation by clearing out unreachable store locations, thus avoiding

future joins which cause imprecision (§ 6.8). This last variation is significant because it demonstrates that even though we have no explicit representation of the stack, it is possible to compute analyses that typically require such explicit representations in order to calculate root sets for garbage collection.

Next, we prove the approach sound w.r.t. a derived big-step collecting and abstract semantics (§ 6.10), where the key insight in the formalism is to model not only standard big-step evaluation relations, but also big-step reachability relations. Finally, we place our work in the context of the prior literature on higher-order abstract interpretation (§ 6.11) and draw some conclusions (§ 6.12).

To convey the ideas of this chapter as concretely as possible, we present code implementing our definitional abstract interpreter and all its variations. As a metalanguage, we use an applicative subset of Racket [Flatt and PLT, 2010], a dialect of Scheme. This choice is largely immaterial: any functional language would do. However, to aide extensibility, we use Racket's *unit* system [Flatt and Felleisen, 1998] to write program components that can be linked together.

6.2 From Machines to Compositional Evaluators

In recent years, there has been considerable effort in the systematic construction of abstract interpreters for higher-order languages using abstract machines—first-order transition systems—as a semantic basis. The so-called *Abstracting Abstract Machines* (AAM) approach to abstract interpretation [Van Horn and Might, 2010] is a recipe for transforming a machine semantics into an easily abstractable form. The

transformation includes the following ingredients:

- Allocating continuations in the store;
- Allocating variable bindings in the store;
- Using a store that maps addresses to *sets* of values;
- Interpreting store updates as a join; and
- Interpreting store dereference as a non-deterministic choice.

These transformations are semantics-preserving due to the original and derived machines operating in a lock-step correspondence. After transforming the semantics in this way, a *computable* abstract interpreter is achieved by:

- Bounding store allocation to a finite set of addresses; and
- Widening base values to some abstract domain.

After performing these transformations, the soundness and computability of the resulting abstract interpreter are then self-evident and easily proved.

The AAM approach has been applied to a wide variety of languages and applications, and given the success of the approach it's natural to wonder what is essential about its use of low-level machines. It is not at all clear whether a similar approach is possible with a higher-level formulation of the semantics, such as a compositional evaluation function defined recursively over the syntax of expressions.

This chapter shows that the essence of the AAM approach can be applied to a high-level semantic basis. We show that compositional evaluators written in monadic

style can express similar abstractions to that of AAM, and like AAM, the design remains systematic. Moreover, we show that the high-level semantics offers a number of benefits not available to the machine model.

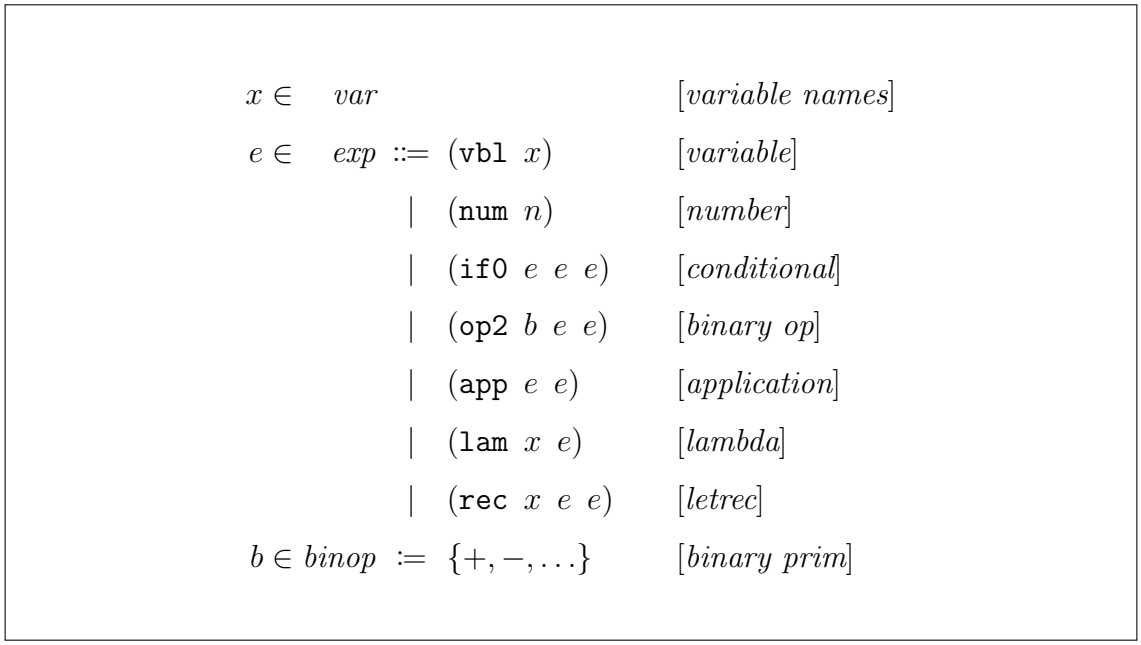
There is a rich body of work concerning tools and techniques for *extensible* interpreters [Jaskelioff, 2009, Kiselyov, 2010, Liang et al., 1995], all of which applies to high-level semantics. By putting abstract interpretation for higher-order languages on a high-level semantic basis, we can bring these results to bear on the construction of extensible abstract interpreters.

6.3 A Definitional Interpreter

We begin by constructing a definitional interpreter for a small but representative higher-order, functional language. The abstract syntax of the language is defined in Figure 6.1; it includes variables, numbers, binary operations on numbers, conditionals, `letrec` expressions, functions and applications.

The interpreter for the language is defined in Figure 6.2. At first glance, it has many conventional aspects:

- It is compositionally defined by structural recursion on the syntax of expressions.
- It represents function values as a closure data structure which pairs the lambda term with the evaluation environment.
- It is structured monadically and uses monad operations to interact with the environment and store.



$$
\begin{array}{lllll}
x \in & \mathit{var} & & & [\mathit{variable\ names}] \\
e \in & \mathit{exp} & ::= & (\texttt{vbl}\ x) & [\mathit{variable}] \\
 & & | & (\texttt{num}\ n) & [\mathit{number}] \\
 & & | & (\texttt{if0}\ e\ e\ e) & [\mathit{conditional}] \\
 & & | & (\texttt{op2}\ b\ e\ e) & [\mathit{binary\ op}] \\
 & & | & (\texttt{app}\ e\ e) & [\mathit{application}] \\
 & & | & (\texttt{lam}\ x\ e) & [\mathit{lambda}] \\
 & & | & (\texttt{rec}\ x\ e\ e) & [\mathit{letrec}] \\
b \in \mathit{binop} & & := & \{+, -, \ldots\} & [\mathit{binary\ prim}]
\end{array}
$$

Figure 6.1: Programming Language Syntax

- It relies on a helper function δ to interpret primitive operations.

There are a few superficial aspects that deserve a quick note: environments ρ are finite maps and the syntax $(\rho\ x)$ denotes $\rho(x)$ while $(\rho\ x\ a)$ denotes $\rho[x \mapsto a]$. The `do`-notation is just shorthand for *bind*, as usual:

$$
\begin{aligned}
(\texttt{do}\ x \leftarrow e\ .\ r) &\equiv (\mathit{bind}\ e\ (\lambda\ (x)\ (\texttt{do}\ .\ r))) \\
(\texttt{do}\ e\ .\ r) &\equiv (\mathit{bind}\ e\ (\lambda\ (\_)\ (\texttt{do}\ .\ r))) \\
(\texttt{do}\ x := e\ .\ r) &\equiv (\texttt{let}\ ((x\ e))\ (\texttt{do}\ .\ r)) \\
(\texttt{do}\ b) &\equiv b
\end{aligned}
$$

Finally, there are two unconventional aspects worth noting.

First, the interpreter is written in an *open recursive style*; the evaluator does not call itself recursively, instead it takes as an argument a function *ev*—shadowing the name of the function *ev* being defined—and *ev* (the argument) is called instead of self-recursion. This is a standard encoding for recursive functions in a setting

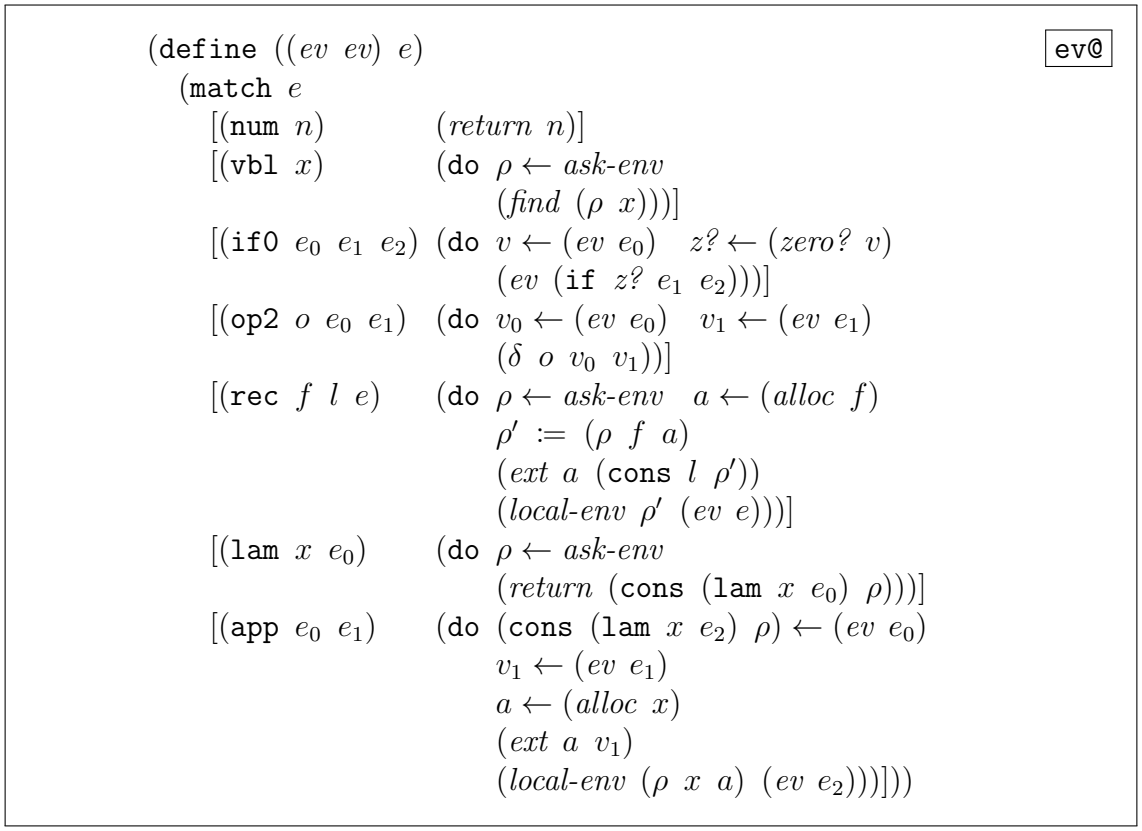


Figure 6.2: The Extensible Definitional Interpreter

without recursive binding. It is up to an external function, such as the Y-combinator, to close the recursive loop. This open recursive form is crucial because it allows intercepting recursive calls to perform "deep" instrumentation of the interpreter.

Second, the code is clearly *incomplete*. There are a number of free variables, typeset as italics, which implement the following:

- The underlying monad of the interpreter: *return* and *bind*;
- An interpretation of primitives: δ and *zero?*;
- Environment operations: *ask-env* for retrieving the environment and *local-env* for installing an environment;
- Store operations: *ext* for updating the store, and *find* for dereferencing locations; and
- An operation for *alloc*ating new store locations.

Going forward, we make frequent use of definitions involving free variables, and we call such a collection of such definitions a *component*. We assume components can be named (in this case, we've named the component `ev@`, indicated by the box in the upper-right corner) and linked together to eliminate free variables. We use Racket *units* [Flatt and Felleisen, 1998] to model components in our implementation.

6.3.1 Instantiating the Interpreter

Next we examine a set of components which complete the definitional interpreter, shown in Figure 6.3. The first component `monad@` uses a macro `define-monad` which

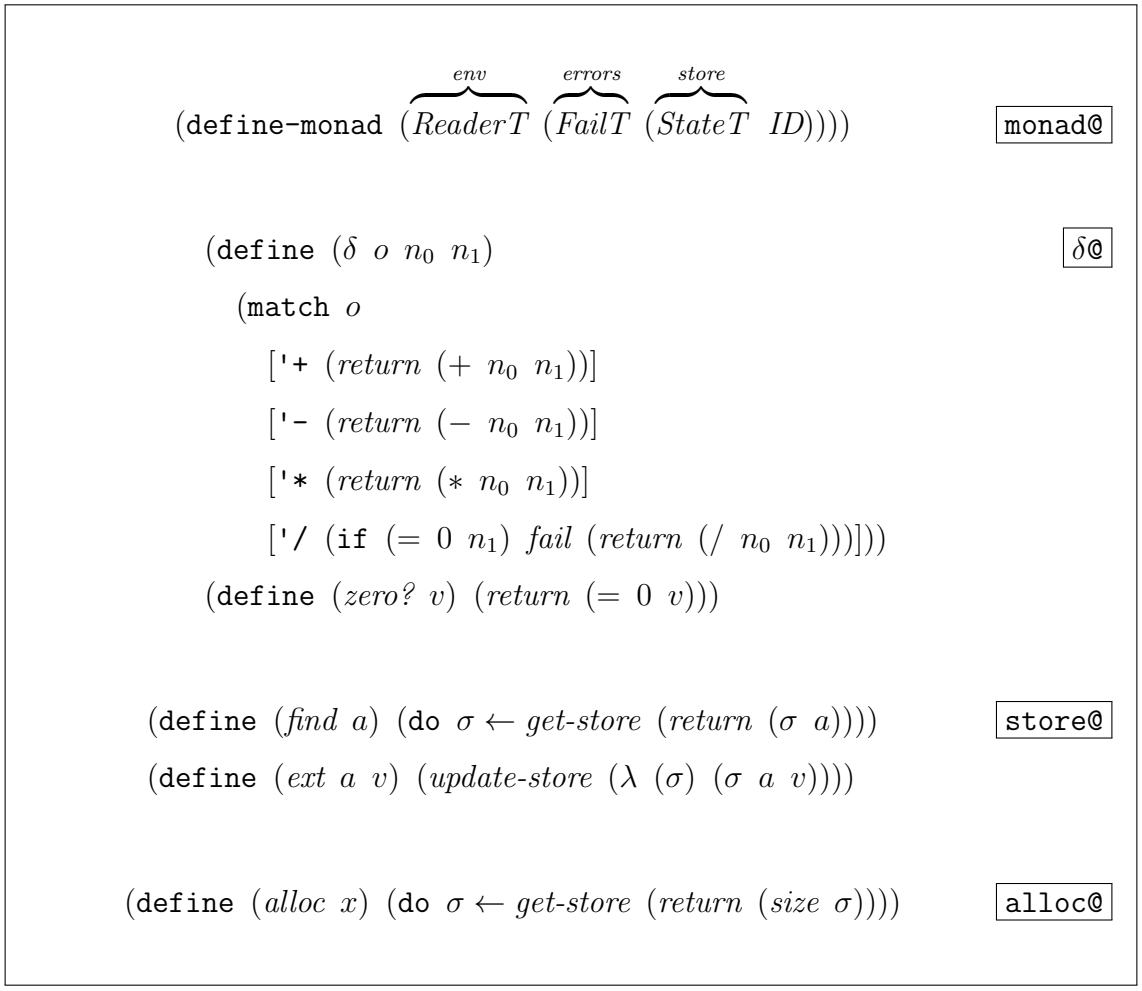


Figure 6.3: Components for Definitional Interpreters

generates a set of bindings based on a monad transformer stack. We use a failure monad to model divide-by-zero errors, a state monad to model the store, and a reader monad to model the environment. The `define-monad` form generates bindings for *return*, *bind*, *ask-env*, *local-env*, *get-store* and *update-store*; their definitions are standard [Liang et al., 1995].

We also define *mrun* for running monadic computations, starting with the

empty environment and store $\varnothing$:

$$(\texttt{define}\ (mrun\ m)\ (run\text{-}StateT\ \varnothing\ (run\text{-}ReaderT\ \varnothing\ m)))$$

While the `define-monad` form is hiding some details, this component could have equivalently been written out explicitly. For example, *return* and *bind* can be defined as:

$$\begin{array}{l}(\texttt{define}\ (((return\ a)\ r)\ s)\ (\texttt{cons}\ a\ s))\\(\texttt{define}\ (((bind\ ma\ f)\ r)\ s)\\\quad(\texttt{match}\ ((ma\ r)\ s)\\\qquad[(\texttt{cons}\ a\ s')\ (((f\ a)\ r)\ s')]\\\qquad[\texttt{'failure}\quad\texttt{'failure}]))\end{array}$$

So far our use of monad transformers is as a mere convenience, however the monad abstraction will become essential for easily deriving new analyses later on.

The δ`@` component defines the interpretation of primitives, which is given in terms of the underlying monad. The `alloc@` component provides a definition of *alloc*, which fetches the store and uses its size to return a fresh address, assuming the invariant $(\in\ a\ \sigma) \Leftrightarrow a < (size\ \sigma)$. The *alloc* function takes a single argument, which is the name of the variable whose binding is being allocated. For the time being, it is ignored, but will become relevant when abstracting closures (§ 6.3.4). The `store@` component defines *find* and *ext* for finding and extending values in the store.

The only remaining pieces of the puzzle are a fixed-point combinator and the

main entry-point for the interpreter, which are straightforward to define:

```
(define ((fix f) x) ((f (fix f)) x))
(define (eval e) (mrun ((fix ev) e)))
```

Using Racket's languages-as-libraries features [Tobin-Hochstadt et al., 2011], we construct REPLs for interacting with this interpreter. Here are a few evaluation examples in a succinct concrete syntax:

```
> (λ (x) x)                                ;; Closure over the empty
'(((λ (x) x) . ()) . ())                   ;;    environment and store.
> ((λ (x) (λ (y) x)) 4)                    ;; Closure over a non-empty
'(((λ (y) x) . ((x . 0))) . ((0 . 4)))     ;;    environment and store.
> (* (+ 3 4) 9)                            ;; Primitive operations work
'(63 . ())                                 ;;    as expected.
> (/ 5 (- 3 3))                            ;; Divide-by-zero errors
'(failure . ())                            ;;    result in failures.
```

Because our monad stack places *FailT* above *StateT*, the answer includes the (empty) store at the point of the error. Had we changed `monad@` to use:

```
(ReaderT (StateT (FailT ID)))
```

then failures would not include the store:

```
> (/ 5 (- 3 3))
'failure
```

At this point we've defined a simple definitional interpreter, although the extensible components involved—monadic operations and open recursion—will allow us to instantiate the same interpreter to achieve a wide range of useful abstract interpretations.

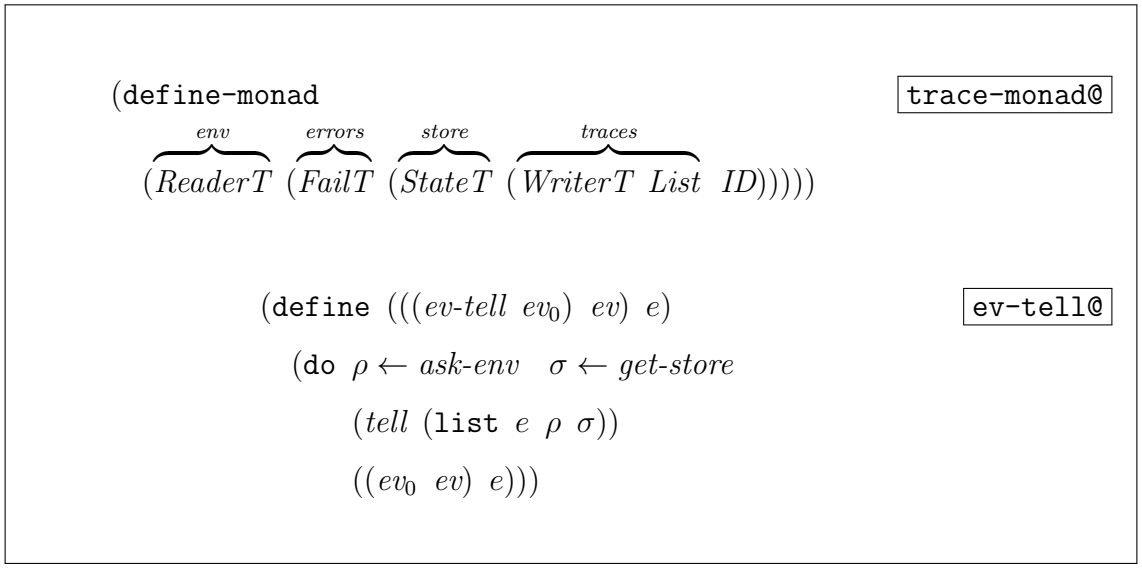


Figure 6.4: Trace Collecting Semantics

6.3.2 Collecting Variations

The formal development of abstract interpretation often starts from a so-called "non-standard collecting semantics." A common form of collecting semantics is a trace semantics, which collects streams of states the interpreter reaches. Figure 6.4 shows the monad stack for a tracing interpreter and a "mix-in" for the evaluator. The monad stack adds *WriterT List*, which provides a new operation *tell* for writing lists of items to the stream of reached states. The *ev-tell* function is a wrapper around an underlying ev_0 unfixed evaluator, and interposes itself between each recursive call by *tell*ing the current state of the evaluator: the current expression, environment and store. The top-level evaluation function is then:

```
(define (eval e) (mrun ((fix (ev-tell ev)) e)))
```

Now when an expression is evaluated, we get an answer and a list of all states

seen by the evaluator, in the order in which they were seen. For example (not showing ρ or σ in results):

```
> (* (+ 3 4) 9)
'((63 . ()) (* (+ 3 4) 9) (+ 3 4) 3 4 9)
> ((λ (x) (λ (y) x)) 4)
'((((λ (y) x) . ((x . 0))) . ((0 . 4)))
  (((λ (x) (λ (y) x)) 4) () ())
  ((λ (x) (λ (y) x)) () ())
  (4 () ())
  ((λ (y) x) ((x . 0)) ((0 . 4))))
```

Were we to swap *List* with *Set* in the monad stack, we would obtain a *reachable* state semantics, another common form of collecting semantics, that loses the order and repetition of states.

As another collecting semantics variant, we show how to collect the *dead code* in a program. Here we use a monad stack that has an additional state component (with operations named *put-dead* and *get-dead*) which stores the set of dead expressions. Initially this will contain all subexpressions of the program. As the interpreter evaluates expressions it will remove them from the dead set.

Figure 6.5 defines the monad stack for the dead code collecting semantics and the `ev-dead@` component, another mix-in for an ev_0 evaluator to remove the given subexpression before recurring. Since computing the dead code requires an outer wrapper that sets the initial set of dead code to be all of the subexpressions in the program, we define `eval-dead@` which consumes a *closed evaluator*, *i.e.* something of the form (*fix ev*). Putting these pieces together, the dead code collecting semantics

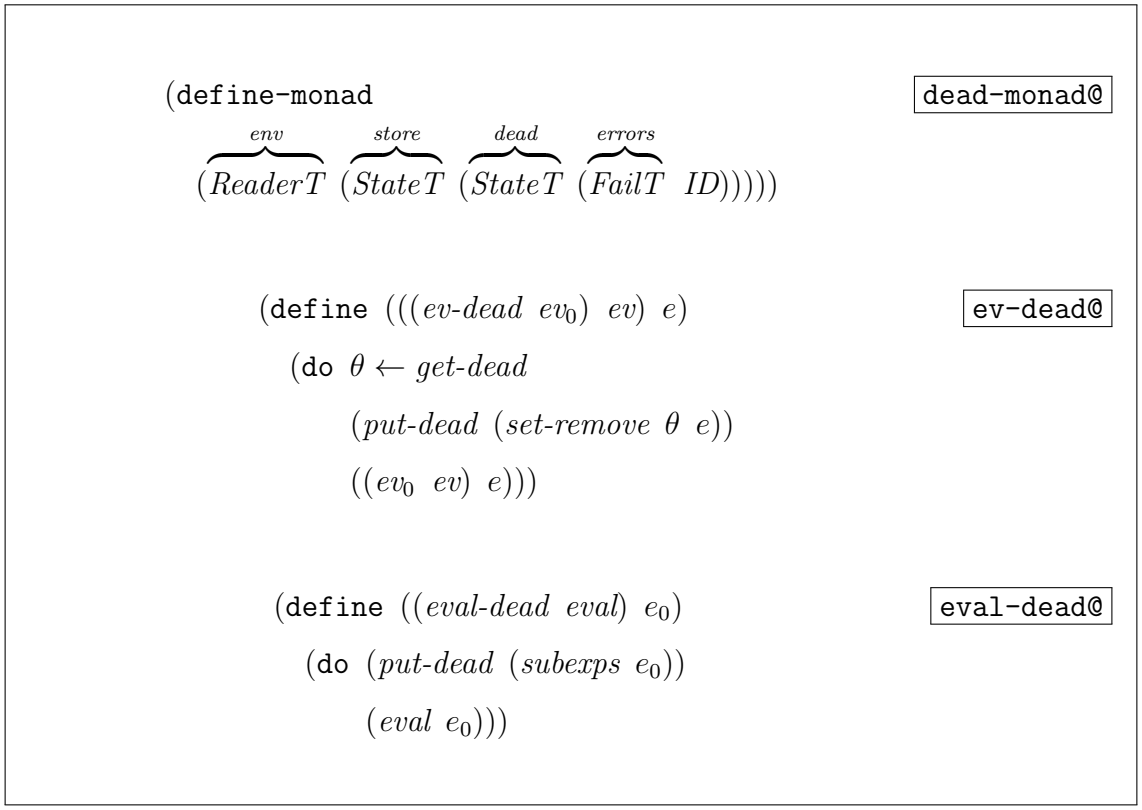


Figure 6.5: Dead Code Collecting Semantics

is defined:

$$(\texttt{define}\ (\mathit{eval}\ e)\ (\mathit{mrun}\ ((\mathit{eval\text{-}dead}\ (\mathit{fix}\ (\mathit{ev\text{-}dead}\ \mathit{ev})))\ e)))$$

Running a program with the dead code interpreter produces an answer and the set of expressions that were not evaluated during the running of a program:

```
> (if0 0 1 2)
(cons '(1 . ()) (set 2))
> (λ (x) x)
(cons '(((λ (x) x) . ()) . ()) (set 'x))
> (if0 (/ 1 0) 2 3)
(cons '(failure . ()) (set 3 2))
```

Our setup makes it easy not only to express the concrete interpreter, but also

```
(define-monad (ReaderT env) (FailT errors) (StateT store) (NondetT mplus) ID)))))   monad^@

(define (δ o n0 n1)                                                                  δ^@
  (match* (o n0 n1)
    [('+ _ _)          (return 'N)]
    [('/ _ (? num?)) (if (= 0 n1) fail (return 'N))]
    [('/ _ 'N)         (mplus fail (return 'N))]
    ...))
(define (zero? v)
  (match v
    ['N (mplus (return #t) (return #f))]
    [_  (return (= 0 v))]))
```

Figure 6.6: Abstracting Primitive Operations

these useful forms of collecting semantics.

6.3.3 Abstracting Base Values

Our interpreter must become decidable before it can be considered an analysis, and the first step towards decidability is to abstract the base types of the language to something finite. We do this for our number base type by introducing a new *abstract* number, written `'N`, which represents the set of all numbers. Abstract numbers are introduced by an alternative interpretation of primitive operations, given in Figure 6.6, which simply produces `'N` in all cases.

Some care must be taken in the abstraction of `'/`. If the denominator is

the abstract number `'N`, then it is possible the program could fail as a result of divide-by-zero, since 0 is contained in the representation of `'N`. Therefore there are *two* possible answers when the denominator is `'N`: `'N` and `'failure`. Both answers are *return*ed by introducing non-determinism $NondetT$ into the monad stack. Adding non-determinism provides the *mplus* operation for combining multiple answers. Non-determinism is also used in *zero?*, which returns both true and false on `'N`.

By linking together `δ^@` and the monad stack with non-determinism, we obtain an evaluator that produces a set of results:

```
> (* (+ 3 4) 9)
'((N . ()))
> (/ 5 (+ 1 2))
'((failure . ()) (N . ()))
> (if0 (+ 1 0) 3 4)
'((3 . ()) (4 . ()))
```

If we link `δ^@` with the *tracing* monad stack plus non-determinism:

$$(\overbrace{ReaderT}^{env}\ (\overbrace{FailT}^{errors}\ (\overbrace{StateT}^{store}\ (\overbrace{WriterT\ List}^{traces}\ (\overbrace{NondetT}^{mplus}\ ID)))))$$

we get an evaluator that produces sets of traces (again not showing ρ or σ in the results):

```
> (if0 (+ 1 0) 3 4)
(set '((3 . ()) (if0 (+ 1 0) 3 4) (+ 1 0) 0 3)
     '((4 . ()) (if0 (+ 1 0) 3 4) (+ 1 0) 0 4))
```

It is clear that the interpreter will only ever see a finite set of numbers (including

`'N`), but it's definitely not true that the interpreter halts on all inputs. First, it's still possible to generate an infinite number of closures. Second, there's no way for the interpreter to detect when it sees a loop. To make a terminating abstract interpreter requires tackling both. We look next at abstracting closures.

6.3.4 Abstracting Closures

Closures consist of code—a lambda term—and an environment—a finite map from variables to addresses. Since the set of lambda terms and variables is bounded by the program text, it suffices to finitize closures by finitizing the set of addresses. Following the AAM approach, we do this by modifying the allocation function to produce elements drawn from a finite set. In order to retain soundness in the semantics, we modify the store to map addresses to *sets* of values, model store update as a join, and model dereference as a non-deterministic choice.

Any abstraction of the allocation function that produces a finite set will do, but the choice of abstraction will determine the precision of the resulting analysis. A simple choice is to allocate variables using the variable's name as its address. This gives a monomorphic, or 0CFA-like, abstraction.

Figure 6.7 shows the component `alloc^@` which implements monomorphic allocation, and the component `store-nd@` for implementing *find* and *ext* which interact with a store mapping to *sets* of values. The `for/monad+` form is a convenience for combining a set of computations with *mplus*, and is used so *find* returns *all* of the values in the store at a given address. The *ext* function joins whenever an address is

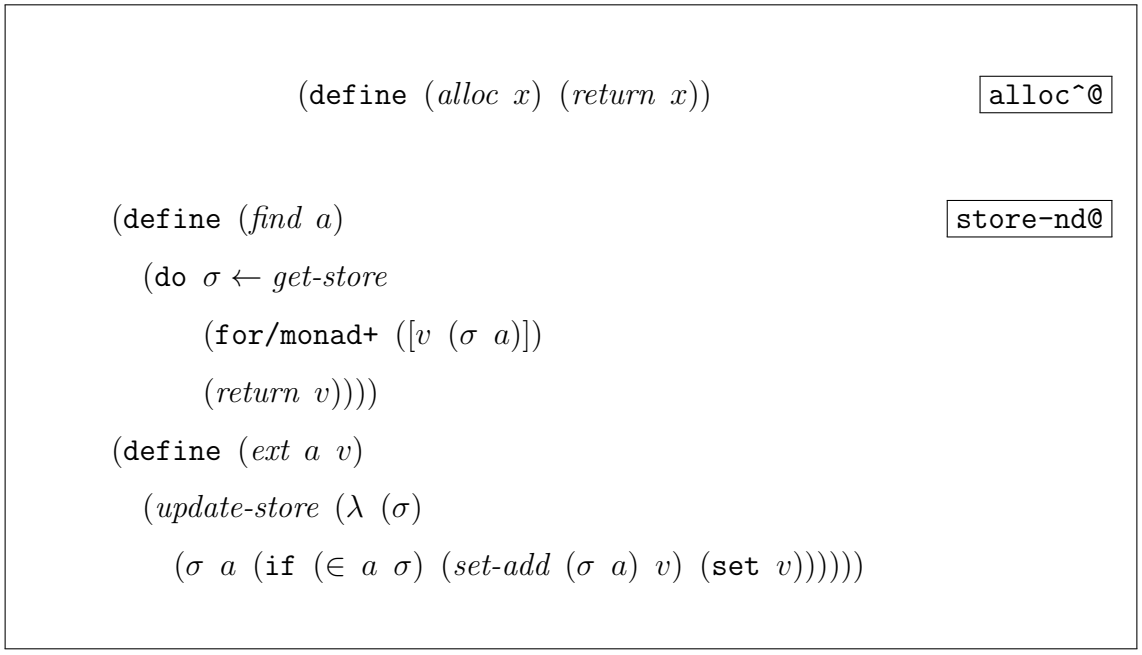


Figure 6.7: Abstracting Allocation: 0CFA

already allocated, otherwise it maps the address to a singleton set. By linking these components with the same monad stack from before, we obtain an interpreter that loses precision whenever variables are bound to multiple values. For example, this program binds x to both 0 and 1 and produces both answers when run:

```
> (let f (λ (x) x)
    (let _ (f 0) (f 1)))]
'((0 . ((x 1 0) (f ((λ (x) x) . ()))))
  (1 . ((x 1 0) (f ((λ (x) x) . ())))))
```

Our abstract interpreter now has a truly finite domain; the next step is to detect loops in the state-space to achieve termination.

6.4 Caching and Finding Fixed-points

At this point, the interpreter obtained by linking together `monad^@`, δ`^@`, `alloc^@` and `store-nd@` components will only ever visit a finite number of configurations for a given program. A configuration (ς) consists of an expression (e), environment (ρ) and store (σ). This configuration is finite because: expressions are finite in the given program; environments are maps from variables (again, finite in the program) to addresses; the addresses are finite thanks to `alloc^`; the store maps addresses to sets of values; base values are abstracted to a finite set by δ`^`; and closures consist of an expression and environment, which are both finite.

Although the interpreter will only ever see a finite set of inputs, it *doesn't know it.* A simple loop will cause the interpreter to diverge:

```
> (rec f (λ (x) (f x)) (f 0))
timeout
```

To solve this problem, we introduce a *cache* ($\$^{in}$) as input to the algorithm, which maps from configurations (ς) to sets of value-and-store pairs ($v \times \sigma$). When a configuration is reached for the second time, rather than re-evaluating the expression and entering an infinite loop, the result is looked up from $\$^{in}$, which acts as an oracle. It is important that the cache is used co-inductively: it is only safe to use $\$^{in}$ as an oracle so long as some progress has been made first.

The results of evaluation are then stored in an output cache ($\$^{out}$), which after the end of evaluation is "more defined" than the input cache ($\$^{in}$), again following a co-inductive argument. The least fixed-point $\$^{+}$ of an evaluator which transforms

an oracle $\$^{in}$ and outputs a more defined oracle $\$^{out}$ is then a sound approximation of the program, because it over-approximates all finite unrollings of the unfixed evaluator.

The co-inductive caching algorithm is shown in Figure 6.8, along with the monad transformer stack `monad-cache@` which has two new components: *ReaderT* for the input cache $\$^{in}$, and *StateT+* for the output cache $\$^{out}$. We use a *StateT+* instead of *WriterT* monad transformer in the output cache so it can double as tracking the set of seen states. The "+" in *StateT+* signifies that caches for multiple non-deterministic branches will be merged automatically, producing a set of results and a single cache, rather than a set of results paired with individual caches.

In the algorithm, when a configuration ς is first encountered, we place an entry in the output cache mapping ς to $(\$^{in}\ \varsigma)$, which is the "oracle" result. Also, whenever we finish computing the result $v \times \sigma'$ of evaluating a configuration ς, we place an entry in the output cache mapping ς to $v \times \sigma'$. Finally, whenever we reach a configuration ς for which a mapping in the output cache exists, we use it immediately, *return*ing each result using the `for/monad+` iterator. Therefore, every "cache hit" on $\$^{out}$ is in one of two possible states: 1) we have already seen the configuration, and the result is the oracle result, as desired; or 2) we have already computed the "improved" result (w.r.t. the oracle), and need not recompute it.

To compute the least fixed-point $\$^{+}$ for the evaluator *ev-cache* we perform a standard Kleene fixed-point iteration starting from the empty map, the bottom element for the cache, as shown in Figure 6.9.

The algorithm runs the caching evaluator *eval* on the given program e from the

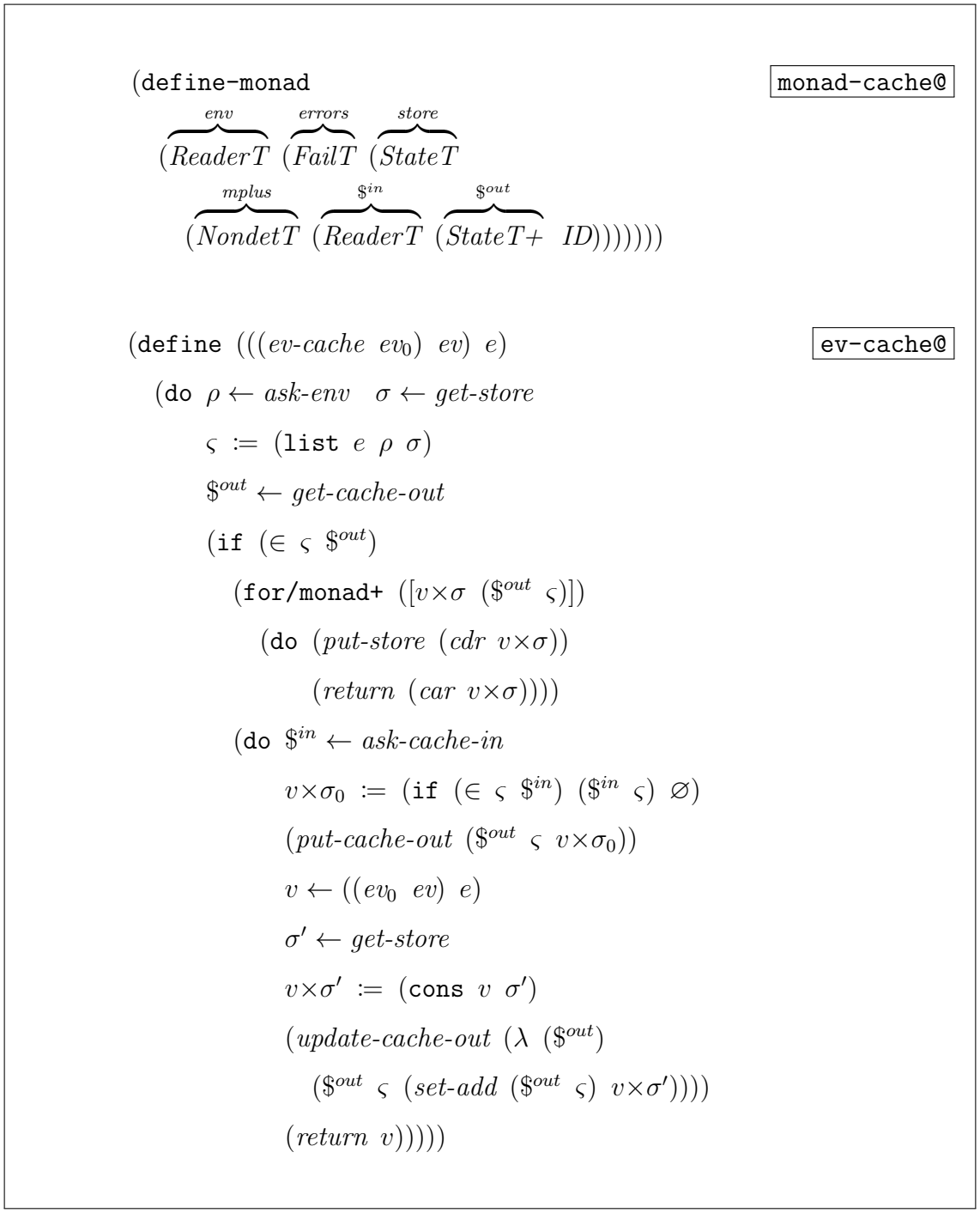


Figure 6.8: Co-inductive Caching Algorithm

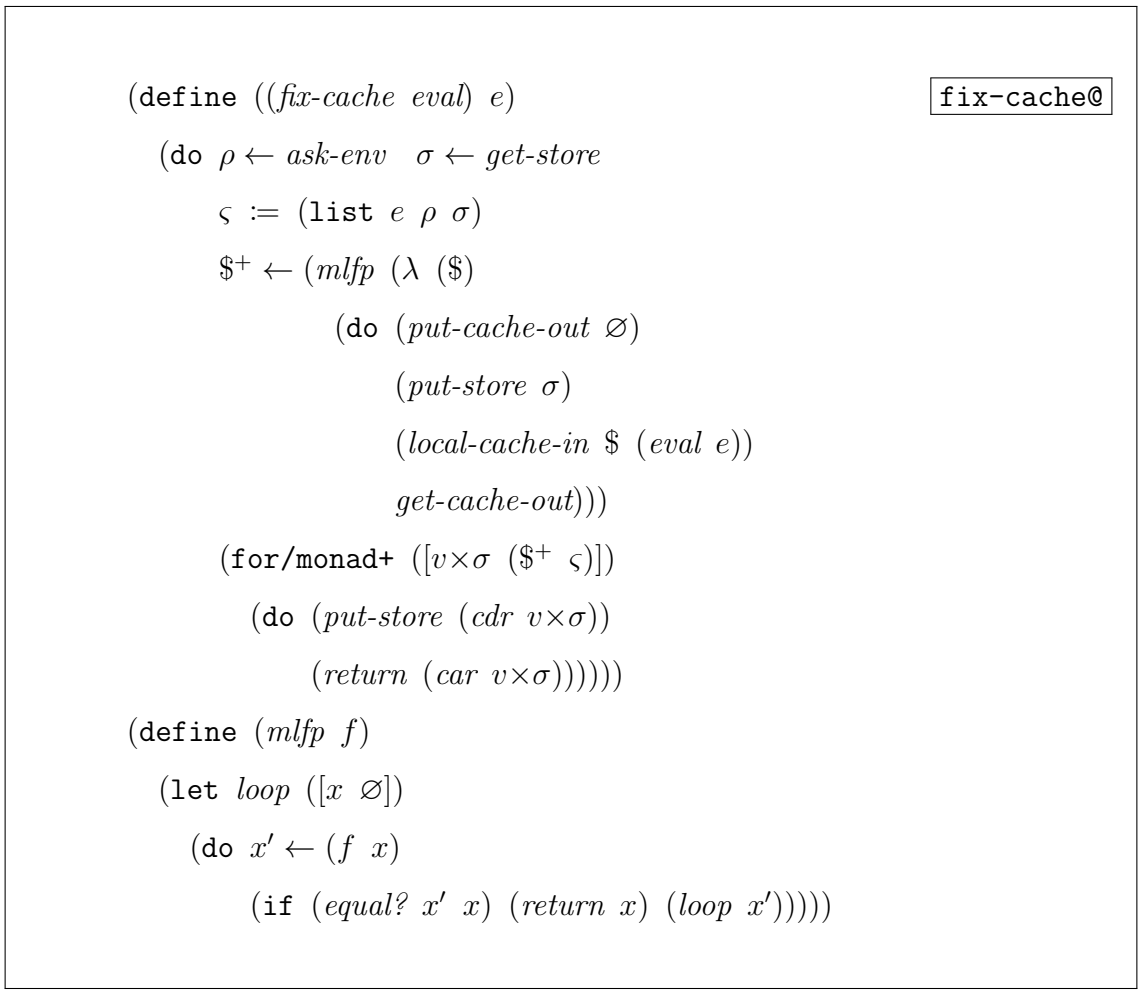


Figure 6.9: Finding Fixed-Points in the Cache

initial environment and store. This is done inside of *mlfp*, a monadic least fixed-point finder. After finding the least fixed-point, the final values and store for the initial configuration ς are extracted and returned.

Termination of the least fixed-point is justified by the monotonicity of the evaluator (it always returns an "improved" oracle), and the finite domain of the cache, which maps abstract configurations to pairs of values and stores, all of which are finite.

With these pieces in place we construct a complete interpreter:

```
(define (eval e) (mrun ((fix-cache (fix (ev-cache ev))) e)))
```

When linked with `δ^` and `alloc^`, this abstract interpreter is sound and computable, as demonstrated on the following examples:

```
> (rec f (λ (x) (f x))
    (f 0))
'()
> (rec f (λ (n) (if0 n 1 (* n (f (- n 1)))))
    (f 5))
'(N)
> (rec f (λ (x) (if0 x 0 (if0 (f (- x 1)) 2 3)))
    (f (+ 1 0)))
'(0 2 3)
```

6.4.1 Formal soundness and termination

In this chapter, we have focused on the code and its intuitions rather than rigorously establishing the usual formal properties of our abstract interpreter, but this is just a matter of presentation: the interpreter is indeed proven sound and computable. We have formalized this co-inductive caching algorithm in Section 6.10, where we prove both that it always terminates, and that it computes a sound over-approximation of concrete evaluation. Here, we give a short summary of our metatheory approach.

In formalising the soundness of this caching algorithm, we extend a standard big-step evaluation semantics into a *big-step reachability* semantics, which characterizes all intermediate configurations which are seen between the evaluation of a single expression and its eventual result. These two notions—*evaluation* which relates expressions to fully evaluated results, and *reachability* which characterizes intermediate configuration states—remain distinct throughout the formalism.

After specifying evaluation and reachability for concrete evaluation, we develop a *collecting* semantics which gives a precise specification for any abstract interpreter, and an *abstract* semantics which partially specifies a sound, over-approximating algorithm w.r.t. the collecting semantics.

The final step is to compute an oracle for the *abstract evaluation relation*, which maps individual configurations to abstractions of the values they evaluate to. To construct this cache, we *mutually* compute the least-fixed point of both the evaluation and reachability relations: based on what is evaluated, discover new things which are reachable, and based on what is reachable, discover new results

of evaluation. The caching algorithm developed in this section is a slightly more efficient strategy for solving the mutual fixed-point, by taking a deep exploration of the reachability relation (up-to seeing the same configuration twice) rather than applying just a single rule of inference.

6.5 Pushdown *à la* Reynolds

By combining the finite abstraction of base values and closures with the termination-guaranteeing cache-based fixed-point algorithm, we have obtained a terminating abstract interpreter. But what kind of abstract interpretation did we get? We have followed the basic recipe of AAM, but adapted to a compositional evaluator instead of an abstract machine. However, we did manage to skip over one of the key steps in the AAM method: we never store-allocated continuations. *In fact, there are no continuations at all.*

A traditional abstract machine formulation of the semantics would model the object-level stack explicitly as an inductively defined data structure. Because stacks may be arbitrarily large, they must be finitized like base values and closures, and like closures, the AAM trick is to thread them through the store, which itself must become finite. But in the definitional interpreter approach, the story of this chapter, the model of the stack is implicit and simply inherited from the meta-language.

But here is the remarkable thing: since the stack is inherited from the meta-language, the abstract interpreter inherits the "call-return matching" of the meta-language, which is to say there is no loss of precision of in the analysis of the

control stack. This is a property that usually comes at considerable effort and engineering in the formulations of higher-order flow analysis that model the stack explicitly. So-called higher-order "pushdown" analysis has been the subject of multiple publications and two dissertations [Earl, 2014, Earl et al., 2010, 2012, Gilray et al., 2016b, Johnson and Van Horn, 2014, Johnson et al., 2014, Van Horn and Might, 2012, Vardoulakis, 2012, Vardoulakis and Shivers, 2011]. Yet when formulated in the definitional interpreter style, the pushdown property requires no mechanics and is simply inherited from the meta-language.

Reynolds, in his celebrated paper *Definitional Interpreters for Higher-order Programming Languages* [Reynolds, 1972], first observed that when the semantics of a programming language is presented as a definitional interpreter, the defined language could inherit semantic properties of the defining metalanguage. We have now shown this observation can be extended to *abstract* interpretation as well, namely in the important case of the pushdown property.

In the remainder of this chapter, we explore a few natural extensions and variations on the basic pushdown abstract interpreter we have established up to this point.

6.6 Widening the Store

In this section, we show how to recover the well-known technique of store-widening in our formulation of a definitional abstract interpreter. This example demonstrates the ease of which we can construct existing abstraction choices.

The abstract interpreter we've constructed so far uses a store-per-program-state abstraction, which is precise but prohibitively expensive. A common technique to combat this cost is to use a global "widened" store [Might, 2007a, Shivers, 1991], which over-approximates each individual store in the current set-up. This change is achieved easily in the monadic setup by re-ordering the monad stack, a technique due to Darais et al. [2015]. Whereas before we had `monad-cache@` we instead swap the order of *StateT* for the store and *NondetT*:

$$\overbrace{(\mathit{ReaderT}}^{\mathit{env}}\ \overbrace{(\mathit{FailT}}^{\mathit{errors}}\ \overbrace{(\mathit{NondetT}}^{\mathit{mplus}}\ \overbrace{(\mathit{StateT}+}^{\mathit{store}}\ \overbrace{(\mathit{ReaderT}}^{\$^{\mathit{in}}}\ \overbrace{(\mathit{StateT}+}^{\$^{\mathit{out}}}\ \mathit{ID})))))$$

we get a store-widened variant of the abstract interpreter. Because *StateT* for the store appears underneath nondeterminism, it will be automatically widened. We write *StateT+* to signify that the cell of state supports such widening. To see the difference, here is an example *without* store-widening:

```
> (let x (+ 1 0)
    (let y (if0 x 1 2)
      (let z (if0 x 3 4)
        (if0 x y z))))
'((4 . ((x N) (y 2) (z 4)))
  (1 . ((x N) (y 1) (z 3)))
  (2 . ((x N) (y 2) (z 3)))
  (3 . ((x N) (y 1) (z 3)))
  (1 . ((x N) (y 1) (z 4)))
  (3 . ((x N) (y 2) (z 3)))
  (2 . ((x N) (y 2) (z 4)))
  (4 . ((x N) (y 1) (z 4))))
```

and an example *with* store-widening:

```
> (let x (+ 1 0)
    (let y (if0 x 1 2)
      (let z (if0 x 3 4)
        (if0 x y z))))
'((1 3 2 4) . ((x N) (y 1 2) (z 3 4)))
```

Notice that before widening, the result is a set of value, store pairs. After widening the result is a pair of a set of values and a store. Importantly, the cache, which bounds the overall run-time of the abstract interpreter, is potentially exponential without store-widening, but collapses to polynomial after store-widening.

6.7 An Alternative Abstraction

In this section, we demonstrate how easy it is to experiment with alternative abstraction strategies by swapping out components. In particular we look at an alternative abstraction of primitive operations and store joins that results in an abstraction that—to the best of our knowledge—has not been explored in the literature. This example shows the potential for rapidly prototyping novel abstractions using our approach.

Figure 6.10 defines two new components: `precise-δ@` and `store-crush@`. The first is an alternative interpretation for primitive operations that is *precision preserving*. Unlike `δ^@`, it does not introduce abstraction, it merely propagates it. When two concrete numbers are added together, the result will be a concrete number, but if either number is abstract then the result is abstract.

precise-δ@

```
(define (δ o n0 n1)
  (match* (o n0 n1)
    [('+ (? num?) (? num?)) (return (+ n0 n1))]
    [('+ _ _)               (return 'N)]
    ...))
(define (zero? v)
  (match v
    ['N (mplus (return #t) (return #f))]
    [_  (return (= 0 v))]))
```

store-crush@

```
(define (find a)
  (do σ ← get-store
      (for/monad+ ([v (σ a)])
        (return v))))
(define (crush v vs)
  (if (closure? v)
      (set-add vs v)
      (set-add (set-filter closure? vs) 'N)))
(define (ext a v)
  (update-store (λ (σ) (if (∈ a σ)
                           (σ a (crush v (σ a)))
                           (σ a (set v))))))
```

Figure 6.10: An Alternative Abstraction for Precise Primitives

This interpretation of primitive operations clearly doesn't impose a finite abstraction on its own, because the state space for concrete numbers is infinite. If `precise-δ@` is linked with the `store-nd@` implementation of the store, termination is therefore not guaranteed.

The `store-crush@` operations are designed to work with `precise-δ@` by performing *widening* when joining multiple concrete values into the store. This abstraction offers a high-level of precision; for example:

```
> (* (+ 3 4) 9)          ;; Constant arithmetic expressions are
'(63)                     ;;     computed with full precision.
> ((λ (x) (* x x)) 5)    ;; Even linear binding and arithmetic
'(25)                     ;;     preserves precision.
> (let f (λ (x) x)       ;; Precision only lost when bindings
    (* (f 5) (f 5)))      ;;     contact base values.
'(N)
```

This combination of `precise-δ@` and `store-crush@` allows termination for most programs, but still not all. In the following example, *id* is eventually applied to a widened argument `'N`, which makes both conditional branches reachable. The function returns 0 in the base case, which is propagated to the recursive call and added to 1, which yields the concrete answer 1. This results in a cycle where the intermediate sum returns 2, 3, 4 when applied to 1, 2, 3, etc.

```
> (rec id (λ (n) (if0 n 0 (+ 1 (id (- n 1)))))
    (id 3))
timeout
```

To ensure termination for all programs, we assume all references to primitive opera-

tions are η-expanded, so that store-allocations also take place at primitive applications, ensuring widening at repeated bindings. In fact, all programs terminate when using `precise-`δ`@`, `store-crush@` and η-expanded primitives, which means we have a achieved a computable and uniquely precise abstract interpreter.

Here we see one of the strengths of the extensible, definitional approach to abstract interpreters. The combination of added precision and widening is encoded quite naturally. In contrast, it's hard to imagine how such a combination could be formulated as, say, a constraint-based flow analysis.

6.8 Symbolic Execution and Garbage Collection

In the published version of this work [Darais et al., 2017] we carry out two examples which demonstrate the wide range of possibilities enabled by the Abstracting Definitional Interpreters technique.

First, we work through an example which shows how to instantiate our definitional abstract interpreter to obtain a symbolic execution engine that performs sound program verification. In the example we describe the monad stack and metafunctions that implement a symbolic executor [King, 1976], then we show how abstractions discussed in previous sections can be applied to enforce termination, turning a traditional symbolic execution into a path-sensitive verification engine.

Next, we show how to incorporate abstract garbage collection [Might and Shivers, 2006a] into our definitional abstract interpreter. The difficulty in defining abstract garbage collection for definitional interpreters is that there is no repre-

sentation of the execution stack to crawl for establishing a root set of reachable addresses. We show how abstract garbage collection can be achieved by extending the instantiated monad with an explicit root set of addresses, and extending the interpreter to perform what looks remarkably similar to off-the-shelf concrete garbage collection.

The result of each of these exercises is the realization that although definitional interpreters defer much of the interpretation structure to the implementing meta-language, complex analysis techniques (like symbolic execution) and introspective analysis techniques (like abstract garbage collection) can still be achieved within the interpreter framework. The key challenges are to (1) achieve the desired semantics through an instrumented definitional interpreter, (2) model the instrumented semantics using new monadic effects as needed, and (3) finitize the state space for the instrumentation.

6.9 Try It Out

All of the components discussed in this chapter have been implemented as units [Flatt and Felleisen, 1998] in Racket [Flatt and PLT, 2010]. We have also implemented a `#lang` language so that composing and experimenting with these interpreters is easy. Assuming Racket is installed, you can install the `monadic-eval` package with:

```
raco pkg install https://github.com/plum-umd/monadic-eval.git
```

A `#lang monadic-eval` program starts with a list of components, which are linked together, and an expression producing an evaluator. Subsequent forms are interpreted as expressions when run. Programs can be run from the command-line or interactively in the DrRacket IDE.

6.10 Formalism

In this section we formalize our approach to designing definitional abstract interpreters. We begin with a "ground truth" big-step semantics and concludes with the fixed-point iteration strategy described in Section 6.4, which we prove sound and computable w.r.t. a synthesized abstract semantics. The design is systematic, and applies to arbitrary developments which use big-step operational semantics. We demonstrate the systematic process as applied to a subset of the language described in Figure 6.1, which we call λ`IF`:

$$
\begin{array}{rrcl}
n \in & \mathbb{N} & & \\
x \in & \mathit{variables} & & \\
b \in & \mathit{binop} & ::= & \{\texttt{plus}, \ldots\} \\
e \in & \mathit{exp} & ::= & n \mid x \mid \lambda x.e \mid e(e) \mid \texttt{if0}(e)\{e\}\{e\} \mid b(e,e) \\
\tau \in & \mathit{time} & ::= & \mathbb{N} \\
\ell \in & \mathit{addr} & ::= & \mathit{var} \times \mathit{time} \\
\rho \in & \mathit{env} & ::= & \mathit{var} \to \mathit{addr}_\bot \\
\sigma \in & \mathit{store} & ::= & \mathit{addr} \to \mathit{val}_\bot \\
v \in & \mathit{val} & ::= & n \mid \langle \lambda x.e, \rho \rangle
\end{array}
$$

Concrete Semantics We begin with the concrete semantics of λ`IF` as a big-step evaluation relation $\rho, \tau \vdash e, \sigma \Downarrow v, \sigma'$, shown in Figure 6.11. The definition is mostly

(Concrete Evaluation) $\boxed{\rho, \tau \vdash e, \sigma \Downarrow v, \sigma'}$

$$(\textsc{Lit})\ \frac{}{\rho, \tau \vdash n, \sigma \Downarrow n, \sigma} \qquad (\textsc{Var})\ \frac{}{\rho, \tau \vdash x, \sigma \Downarrow \sigma(\rho(x)), \sigma}$$

$$(\textsc{Lam})\ \frac{}{\rho, \tau \vdash \lambda x.e, \sigma \Downarrow \langle \lambda x.e, \rho \rangle, \sigma}$$

$$(\textsc{Bin})\ \frac{\rho, \tau \vdash e_1, \sigma \Downarrow v_1, \sigma_1 \qquad \rho, \tau \vdash e_2, \sigma_1 \Downarrow v_2, \sigma_2}{\rho, \tau \vdash b(e_1, e_2), \sigma \Downarrow [\![b]\!](v_1, v_2), \sigma_2}$$

$$(\textsc{App})\ \frac{\rho, \tau \vdash e_1, \sigma \Downarrow v_1, \sigma_1 \qquad \rho, \tau \vdash e_2, \sigma_1 \Downarrow v_2, \sigma_2 \qquad \rho'[x \mapsto \ell], \tau' \vdash e', \sigma_2[\ell \mapsto v_2] \Downarrow v', \sigma_3}{\rho, \tau \vdash e_1(e_2), \sigma \Downarrow v', \sigma_3} \begin{array}{l} \langle \lambda x.e', \rho' \rangle = v_1 \\ \ell = \langle x, \tau' \rangle \\ \tau' \quad \textit{fresh} \end{array}$$

$$(\textsc{IfT})\ \frac{\rho, \tau \vdash e_1, \sigma \Downarrow n, \sigma_1 \qquad \rho, \tau \vdash e_2, \sigma_1 \Downarrow v, \sigma_2}{\rho, \tau \vdash \texttt{if0}(e_1)\{e_2\}\{e_3\}, \sigma \Downarrow v, \sigma_2} n = 0$$

$$(\textsc{IfF})\ \frac{\rho, \tau \vdash e_1, \sigma \Downarrow n, \sigma_1 \qquad \rho, \tau \vdash e_3, \sigma_1 \Downarrow v, \sigma_2}{\rho, \tau \vdash \texttt{if0}(e_1)\{e_2\}\{e_3\}, \sigma \Downarrow v, \sigma_2} n \neq 0$$

Figure 6.11: λIF Big-step Concrete Evaluation Semantics

standard: ρ and σ are the environment and store, e is the initial expression, and v is the resulting value. The argument τ represents "time," which when abstracted supports modeling execution contexts like call-site sensitivity. Concretely time is modeled as a natural number, and all that is required is that "fresh" numbers are available for allocating values in the store.

REACHABILITY The primary limitation of using big-step semantics as a starting point for abstraction is that intermediate computations are not represented in the model for evaluation. For example, consider the program that applies the identity function to an expression that loops, which we notate Ω:

$$(\lambda x.x)(\Omega)$$

A big-step evaluation relation can only describe results of terminating computations, and because this program never terminates, such a relation says nothing about the behavior of the program. A good static analyzer will explore the behavior of Ω to (possibly) discover that it loops, or more importantly, to provide analysis results (like data-flow or side-effects) for intermediate computation states.

The need to analyze intermediate states is the primary reason that big-step semantics are overlooked as a starting point for abstract interpretation. To remedy the situation, while remaining in a big-step setting, we introduce a big-step *reachability* relation, notated $\rho, \tau \vdash e, \sigma \Uparrow \varsigma$ and shown in Figure 6.12. Configurations ς are tuples $\langle e, \rho, \sigma, \tau \rangle$, and are reachable when evaluation passes through the configuration at any point on its way to a final value, or during an infinite loop.

The complete big-step semantics of an expression (e) under environment (ρ), store (σ) and time (τ), which we notate $[\![e]\!]^{bs}(\rho, \sigma, \tau)$, is then the set of all *reachable evaluations*:

$$\begin{aligned}[\![e]\!]^{bs}(\rho, \sigma, \tau) \coloneqq \{\langle v, \sigma'' \rangle \mid \rho, \sigma, \tau \Uparrow \langle e', \rho', \sigma', \tau' \rangle \\ \wedge \rho', \tau' \vdash e', \sigma' \Downarrow v, \sigma''\}\end{aligned}$$

We construct a formal bridge between the big-step and small-step worlds through

(Concrete Reachability) $\boxed{\rho, \tau \vdash e, \sigma \Uparrow \varsigma}$

$$(\textsc{Refl})\frac{}{\rho, \tau \vdash e, \sigma \Uparrow \langle e, \rho, \sigma, \tau \rangle} \qquad (\textsc{RBin1})\frac{\rho, \tau \vdash e_1, \sigma \Uparrow \varsigma}{\rho, \tau \vdash b(e_1, e_2), \sigma \Uparrow \varsigma}$$

$$(\textsc{RBin2})\frac{\rho, \tau \vdash e_1, \sigma \Downarrow v_1, \sigma_1 \qquad \rho, \tau \vdash e_2, \sigma_1 \Uparrow \varsigma}{\rho, \tau \vdash b(e_1, e_2), \sigma \Uparrow \varsigma}$$

$$(\textsc{RApp1})\frac{\rho, \tau \vdash e_1, \sigma \Uparrow \varsigma}{\rho, \tau \vdash e_1(e_2), \sigma \Uparrow \varsigma}$$

$$(\textsc{RApp2})\frac{\rho, \tau \vdash e_1, \sigma \Downarrow v_1, \sigma_1 \rho, \tau \vdash e_2, \sigma_1 \Uparrow \varsigma}{\rho, \tau \vdash e_1(e_2), \sigma \Uparrow \varsigma} \langle \lambda x.e', \rho' \rangle = v_1$$

$$(\textsc{RApp3})\frac{\begin{array}{c}\rho, \tau \vdash e_1, \sigma \Downarrow v_1, \sigma_1 \qquad \rho, \tau \vdash e_2, \sigma_1 \Downarrow v_2, \sigma_2 \\ \rho'[x \mapsto \ell], \tau' \vdash e', \sigma_2[\ell \mapsto v_2] \Uparrow \varsigma\end{array}}{\rho, \tau \vdash e_1(e_2), \sigma \Uparrow \varsigma} \begin{array}{r}\langle \lambda x.e', \rho' \rangle = v_1 \\ \ell = \langle x, \tau' \rangle \\ \tau' \quad \textit{fresh}\end{array}$$

$$(\textsc{RIf1})\frac{\rho, \tau \vdash e_1, \sigma \Uparrow \varsigma}{\rho, \tau \vdash \texttt{if0}(e_1)\{e_2\}\{e_3\}, \sigma \Uparrow \varsigma}$$

$$(\textsc{RIfT})\frac{\rho, \tau \vdash e_1, \sigma \Downarrow n, \sigma_1 \qquad \rho, \tau \vdash e_2, \sigma_1 \Uparrow \varsigma}{\rho, \tau \vdash \texttt{if0}(e_1)\{e_2\}\{e_3\}, \sigma \Uparrow \varsigma} n = 0$$

$$(\textsc{RIfF})\frac{\rho, \tau \vdash e_1, \sigma \Downarrow n, \sigma_1 \qquad \rho, \tau \vdash e_3, \sigma_1 \Uparrow \varsigma}{\rho, \tau \vdash \texttt{if0}(e_1)\{e_2\}\{e_3\}, \sigma \Uparrow \varsigma} n \neq 0$$

Figure 6.12: λIF Big-step Concrete Reachability Semantics

the complete big-step semantics ($[\![e]\!]^{bs}$) and a complete small-step semantics $\rightsquigarrow^*$, which is traditionally used as the starting point of abstraction for program analysis:

$$\begin{aligned}[\![e]\!]^{ss}(\rho,\sigma,\tau) \;\coloneqq\; \{\langle v,\sigma''\rangle \mid \forall\kappa.\; \langle e,\rho,\sigma,\tau,\kappa\rangle \rightsquigarrow^* \langle e',\rho',\sigma',\tau',\kappa' +\!\!+ \kappa\rangle \\ \wedge\, \langle e',\rho',\sigma',\tau',\kappa' +\!\!+ \kappa\rangle \rightsquigarrow^* \langle v,\rho'',\sigma'',\tau'',\kappa' +\!\!+ \kappa\rangle\}\end{aligned}$$

We connect the complete big-step and small-step semantics through the following theorem:

Theorem 7 (Complete Big-step/Small-step Equivalence)**.**

$$[\![e]\!]^{bs}(\rho,\sigma,\tau) = [\![e]\!]^{ss}(\rho,\sigma,\tau)$$

The proof is by induction on the big-step derivation for $\subseteq$, and on the transitive small-step derivation for $\supseteq$.

Collecting Semantics Before abstracting the semantics—in pursuit of a sound static analysis algorithm—we pass through a big-step collecting evaluation and reachability semantics, notated $\rho,\tau \vdash e,\widetilde{\sigma} \Downarrow \widetilde{v},\widetilde{\sigma}$ and $\rho,\tau \vdash e,\widetilde{\sigma} \Uparrow \widetilde{\varsigma}$ and shown in figures 6.13 and 6.14, where $\widetilde{v}$, $\widetilde{\sigma}$ and $\widetilde{\varsigma}$ range over collecting state spaces:

$$\begin{aligned}\widetilde{v} \in \quad \widetilde{val} &\coloneqq \wp(val)\\ \widetilde{\sigma} \in \widetilde{store} &\coloneqq addr \mapsto \widetilde{val}\\ \widetilde{\varsigma} \in \widetilde{config} &\coloneqq exp \times env \times \widetilde{store} \times time\end{aligned}$$

and the denotation for binary operators ($[\![b]\!]$) is lifted to a collecting denotation operator $\widetilde{[\![b]\!]}$:

$$\widetilde{[\![b]\!]}(\widetilde{v}_1,\widetilde{v}_2) \;\coloneqq\; \{[\![b]\!](v_1,v_2) \mid v_1 \in \widetilde{v}_1 \wedge v_2 \in \widetilde{v}_2\}$$

The big-step collecting and reachability relations are structurally similar to the

(Collecting Evaluation) $\boxed{\rho, \tau \vdash e, \widetilde{\sigma} \Downarrow \widetilde{v}, \widetilde{\sigma}'}$

$$(\textsc{OLit})\ \frac{}{\rho, \tau \vdash n, \widetilde{\sigma} \Downarrow \{n\}, \widetilde{\sigma}} \qquad (\textsc{OVar})\ \frac{}{\rho, \tau \vdash x, \widetilde{\sigma} \Downarrow \widetilde{\sigma}(\rho(x)), \widetilde{\sigma}}$$

$$(\textsc{OLam})\ \frac{}{\rho, \tau \vdash \lambda x.e, \widetilde{\sigma} \Downarrow \{\langle \lambda x.e, \rho \rangle\}, \widetilde{\sigma}}$$

$$(\textsc{OBin})\ \frac{\rho, \tau \vdash e_1, \widetilde{\sigma} \Downarrow \widetilde{v}_1, \widetilde{\sigma}_1 \qquad \rho, \tau \vdash e_2, \widetilde{\sigma}_1 \Downarrow \widetilde{v}_2, \widetilde{\sigma}_2}{\rho, \tau \vdash b(e_1, e_2), \widetilde{\sigma} \Downarrow \widetilde{[\![b]\!]}(\widetilde{v}_1, \widetilde{v}_2), \widetilde{\sigma}_2}$$

$$(\textsc{OApp})\ \frac{\begin{array}{c}\rho, \tau \vdash e_1, \widetilde{\sigma} \Downarrow \widetilde{v}_1, \widetilde{\sigma}_1 \qquad \rho, \tau \vdash e_2, \widetilde{\sigma}_1 \Downarrow \widetilde{v}_2, \widetilde{\sigma}_2 \\ \rho'[x \mapsto \ell], \tau' \vdash e', \widetilde{\sigma}_2[\ell \mapsto \widetilde{v}_2] \Downarrow \widetilde{v}', \widetilde{\sigma}_3\end{array}}{\rho, \tau \vdash e_1(e_2), \widetilde{\sigma} \Downarrow \widetilde{v}', \widetilde{\sigma}_3} \begin{array}{l}\langle \lambda x.e', \rho' \rangle \in \widetilde{v}_1 \\ \ell = \langle x, \tau' \rangle \\ \tau' \quad \textit{fresh}\end{array}$$

$$(\textsc{OIfT})\ \frac{\rho, \tau \vdash e_1, \widetilde{\sigma} \Downarrow \widetilde{v}_1, \widetilde{\sigma}_1 \qquad \rho, \tau \vdash e_2, \widetilde{\sigma}_1 \Downarrow \widetilde{v}, \widetilde{\sigma}_2}{\rho, \tau \vdash \texttt{if0}(e_1)\{e_2\}\{e_3\}, \widetilde{\sigma} \Downarrow \widetilde{v}, \widetilde{\sigma}_2} 0 \in \widetilde{v}_1$$

$$(\textsc{OIfF})\ \frac{\rho, \tau \vdash e_1, \widetilde{\sigma} \Downarrow \widetilde{v}_1, \widetilde{\sigma}_1 \qquad \rho, \tau \vdash e_3, \widetilde{\sigma}_1 \Downarrow \widetilde{v}, \widetilde{\sigma}_2}{\rho, \tau \vdash \texttt{if0}(e_1)\{e_2\}\{e_3\}, \widetilde{\sigma} \Downarrow \widetilde{v}, \widetilde{\sigma}_2} \begin{array}{l} n \in \widetilde{v}_1 \\ n \neq 0 \end{array}$$

Figure 6.13: Big-step Collecting Evaluation Semantics

(Collecting Reachability) $\boxed{\rho, \tau \vdash e, \widetilde{\sigma} \Uparrow \widetilde{\varsigma}}$

$$(\text{OREFL})\frac{}{\rho, \tau \vdash e, \widetilde{\sigma} \Uparrow \langle e, \rho, \widetilde{\sigma}, \tau \rangle} \qquad (\text{ORBIN1})\frac{\rho, \tau \vdash e_1, \widetilde{\sigma} \Uparrow \varsigma}{\rho, \tau \vdash b(e_1, e_2), \widetilde{\sigma} \Uparrow \widetilde{\varsigma}}$$

$$(\text{ORBIN2})\frac{\rho, \tau \vdash e_1, \widetilde{\sigma} \Downarrow \widetilde{v}_1, \widetilde{\sigma}_1 \qquad \rho, \tau \vdash e_2, \widetilde{\sigma}_1 \Uparrow \widetilde{\varsigma}}{\rho, \tau \vdash b(e_1, e_2), \widetilde{\sigma} \Uparrow \widetilde{\varsigma}}$$

$$(\text{ORAPP1})\frac{\rho, \tau \vdash e_1, \widetilde{\sigma} \Uparrow \widetilde{\varsigma}}{\rho, \tau \vdash e_1(e_2), \widetilde{\sigma} \Uparrow \widetilde{\varsigma}}$$

$$(\text{ORAPP2})\frac{\rho, \tau \vdash e_1, \widetilde{\sigma} \Downarrow \widetilde{v}_1, \widetilde{\sigma}_1 \qquad \rho, \tau \vdash e_2, \widetilde{\sigma}_1 \Uparrow \widetilde{\varsigma}}{\rho, \tau \vdash e_1(e_2), \widetilde{\sigma} \Uparrow \widetilde{\varsigma}} \lambda x.e', \rho' \rangle \in \widetilde{v}_1$$

$$(\text{ORAPP3})\frac{\begin{array}{c}\rho, \tau \vdash e_1, \widetilde{\sigma} \Downarrow \widetilde{v}_1, \widetilde{\sigma}_1 \qquad \rho, \tau \vdash e_2, \widetilde{\sigma}_1 \Downarrow \widetilde{v}_2, \widetilde{\sigma}_2 \\ \rho'[x \mapsto \ell], \tau' \vdash e', \widetilde{\sigma}_2[\ell \mapsto \widetilde{v}_2] \Uparrow \widetilde{\varsigma}\end{array}}{\rho, \tau \vdash e_1(e_2), \widetilde{\sigma} \Uparrow \widetilde{\varsigma}} \begin{array}{l}\langle \lambda x.e', \rho' \rangle \in \widetilde{v}_1 \\ \ell = \langle x, \tau' \rangle \\ \tau' \quad \textit{fresh}\end{array}$$

$$(\text{ORIF1})\frac{\rho, \tau \vdash e_1, \widetilde{\sigma} \Uparrow \widetilde{\varsigma}}{\rho, \tau \vdash \texttt{if0}(e_1)\{e_2\}\{e_3\}, \widetilde{\sigma} \Uparrow \widetilde{\varsigma}}$$

$$(\text{ORIFT})\frac{\rho, \tau \vdash e_1, \widetilde{\sigma} \Downarrow \widetilde{v}_1, \widetilde{\sigma}_1 \qquad \rho, \tau \vdash e_2, \widetilde{\sigma}_1 \Uparrow \widetilde{\varsigma}}{\rho, \tau \vdash \texttt{if0}(e_1)\{e_2\}\{e_3\}, \widetilde{\sigma} \Uparrow \widetilde{\varsigma}} 0 \in \widetilde{v}_1$$

$$(\text{ORIFF})\frac{\rho, \tau \vdash e_1, \widetilde{\sigma} \Downarrow \widetilde{v}_1, \widetilde{\sigma}_1 \qquad \rho, \tau \vdash e_3, \widetilde{\sigma}_1 \Uparrow \widetilde{\varsigma}}{\rho, \tau \vdash \texttt{if0}(e_1)\{e_2\}\{e_3\}, \widetilde{\sigma} \Uparrow \widetilde{\varsigma}} \begin{array}{l} n \in \widetilde{v}_1 \\ n \neq 0 \end{array}$$

Figure 6.14: Big-step Collecting Reachability Semantics

concrete semantics. The primary differences are the use of set containment ($\in$) in place of equality ($=$) when branching on application and conditional expressions.

The big-step collecting reachability semantics is a sound approximation of the big-step concrete reachability semantics:

Theorem 8 (Collecting Reachability Semantics Soundness)**.**

$$\begin{array}{rl} \textit{If} & \rho, \tau \vdash e, \sigma \Uparrow \langle e', \rho', \sigma', \tau' \rangle \quad \textit{and} \quad \rho', \tau' \vdash e', \sigma' \Downarrow v, \sigma'' \\ \textit{where} & \eta(\sigma) \sqsubseteq \widetilde{\sigma} \\ \textit{then} & \rho, \tau \vdash e, \widetilde{\sigma} \Uparrow \langle e', \rho', \widetilde{\sigma}', \tau' \rangle \quad \textit{and} \quad \rho', \tau' \vdash e, \widetilde{\sigma}' \Downarrow \widetilde{v}, \widetilde{\sigma}'' \\ \textit{where} & \eta(\sigma') \sqsubseteq \widetilde{\sigma}' \quad \textit{and} \quad v \in \widetilde{v} \quad \textit{and} \quad \eta(\sigma'') \sqsubseteq \widetilde{\sigma}'' \end{array}$$

The proof is by induction on the concrete big-step derivation. The extraction function η is defined separately for stores (σ) and configurations (ς):

$$\eta(\sigma)(\ell) \coloneqq \{\sigma(\ell)\} \qquad\qquad \eta(\langle e, \rho, \sigma, \tau \rangle) \coloneqq \langle e, \rho, \eta(\sigma), \tau \rangle$$

and the partial ordering on stores and configurations is pointwise:

$$\begin{array}{c} \widetilde{\sigma}_1 \sqsubseteq \widetilde{\sigma}_2 \quad \textit{iff} \quad \forall \ell.\ \widetilde{\sigma}_1(\ell) \subseteq \widetilde{\sigma}_2(\ell) \\ \langle e_1, \rho_1, \widetilde{\sigma}_1, \tau_1 \rangle \sqsubseteq \langle e_2, \rho_2, \widetilde{\sigma}_2, \tau_2 \rangle \quad \textit{iff} \quad e_1 = e_2 \wedge \rho_1 = \rho_2 \wedge \widetilde{\sigma}_1 \sqsubseteq \widetilde{\sigma}_2 \wedge \tau_1 = \tau_2 \end{array}$$

Finite Abstraction The next step towards a computable static analysis is an abstract semantics with a finite state space that approximates the big-step collecting semantics, notated $\widehat{\rho}, \widehat{\tau} \vdash e, \widehat{\sigma} \Downarrow \widehat{v}, \widehat{\sigma}$ and $\widehat{\rho}, \widehat{\tau} \vdash e, \widehat{\sigma} \Uparrow \widehat{\varsigma}$ and shown in figures 6.15

and 6.16, where $\widehat{\rho}$, $\widehat{\tau}$, $\widehat{v}$, $\widehat{\sigma}$ and $\widehat{\varsigma}$ are finite abstractions of their collecting counterparts:

$$\begin{array}{rrcl}
\widehat{\rho} \in & \widehat{env} & \coloneqq & var \mapsto \widehat{addr}_\perp \\
\widehat{\ell} \in & \widehat{addr} & \coloneqq & var \times \widehat{time} \\
\widehat{\tau} \in & \widehat{time} & \coloneqq & \ldots \\
\widehat{v} \in & \widehat{val} & \coloneqq & \ldots \\
\widehat{\sigma} \in & \widehat{store} & \coloneqq & \widehat{addr} \mapsto \widehat{val} \\
\widehat{\varsigma} \in & \widehat{config} & \coloneqq & exp \times \widehat{env} \times \widehat{store} \times \widehat{time}
\end{array}$$

The primary structural difference from the collecting semantics is the use of join when updating the store ($\widehat{\sigma} \sqcup [\widehat{\ell} \mapsto \widehat{v}]$) rather than strict replacement ($\widetilde{\sigma}[\ell \mapsto \widetilde{v}]$). This is to preserve soundness in the presence of address reuse, which occurs from the finite size of the address space.

The abstract denotation ($\widehat{[\![b]\!]}$) is any over-approximation of the collecting denotation ($\widetilde{[\![b]\!]}$) w.r.t. a Galois connection $\widetilde{val} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \widehat{val}$:

$$\widehat{[\![b]\!]}(\widehat{v}_1, \widehat{v}_2) \sqsupseteq \alpha(\widetilde{[\![b]\!]}(\gamma(\widehat{v}_1), \gamma(\widehat{v}_2)))$$

Concretization functions $\lfloor\gamma\rfloor_{clo}$, $\lfloor\gamma\rfloor_0$ and $\lfloor\gamma\rfloor_{\neg 0}$ are computable finite subsets of the full concretization function γ s.t.:

$$\begin{array}{rcl}
\lfloor\gamma\rfloor_{clo}(\widehat{v}) & \coloneqq & \{\langle \lambda x.e, \widehat{\rho}\rangle \mid \langle \lambda x.e, \widehat{\rho}\rangle \in \gamma(\widehat{v})\} \\
\lfloor\gamma\rfloor_0(\widehat{v}) & \coloneqq & \{0 \mid 0 \in \gamma(\widehat{v})\} \\
\lfloor\gamma\rfloor_{\neg 0}(\widehat{v}) & \coloneqq & \{\neg 0 \mid n \in \gamma(\widehat{v}) \wedge n \neq 0\}
\end{array}$$

Abstract sets $\widehat{time}$ and $\widehat{val}$ are left as parameters to the analysis along with their operations $\widehat{next}$, $\widehat{[\![b]\!]}$, $\lfloor\gamma\rfloor_{clo}$, $\lfloor\gamma\rfloor_0$, $\lfloor\gamma\rfloor_{\neg 0}$ and $\sqcup^{\widehat{val}}$.

The abstract semantics is a sound approximation of the collecting semantics, which we establish through the theorem:

(Abstract Evaluation) $\boxed{\widehat{\rho}, \widehat{\tau} \vdash e, \widehat{\sigma} \Downarrow \widehat{v}, \widehat{\sigma}'}$

$$(\textsc{ALit})\frac{}{\widehat{\rho}, \widehat{\tau} \vdash n, \widehat{\sigma} \Downarrow \widehat{\eta}(n), \widehat{\sigma}} \qquad (\textsc{AVar})\frac{}{\widehat{\rho}, \widehat{\tau} \vdash x, \widehat{\sigma} \Downarrow \widehat{\sigma}(\widehat{\rho}(x)), \widehat{\sigma}}$$

$$(\textsc{ALam})\frac{}{\widehat{\rho}, \widehat{\tau} \vdash \lambda x.e, \widehat{\sigma} \Downarrow \widehat{\eta}(\langle \lambda x.e, \widehat{\rho} \rangle), \widehat{\sigma}}$$

$$(\textsc{ABin})\frac{\widehat{\rho}, \widehat{\tau} \vdash e_1, \widehat{\sigma} \Downarrow \widehat{v}_1, \widehat{\sigma}_1 \qquad \widehat{\rho}, \widehat{\tau} \vdash e_2, \widehat{\sigma}_1 \Downarrow \widehat{v}_2, \widehat{\sigma}_2}{\widehat{\rho}, \widehat{\tau} \vdash b(e_1, e_2), \widehat{\sigma} \Downarrow \widehat{[\![b]\!]}(\widehat{v}_1, \widehat{v}_2), \widehat{\sigma}_2}$$

$$(\textsc{AApp})\frac{\begin{array}{c}\widehat{\rho}, \widehat{\tau} \vdash e_1, \widehat{\sigma} \Downarrow \widehat{v}_1, \widehat{\sigma}_1 \qquad \widehat{\rho}, \widehat{\tau} \vdash e_2, \widehat{\sigma}_1 \Downarrow \widehat{v}_2, \widehat{\sigma}_2 \quad \langle \lambda x.e', \widehat{\rho}' \rangle \in \lfloor \gamma \rfloor_{clo}(\widehat{v}_1) \\ \widehat{\rho}'[x \mapsto \widehat{\ell}], \widehat{\tau}' \vdash e', \widehat{\sigma}_2 \sqcup [\widehat{\ell} \mapsto \widehat{v}_2] \Downarrow \widehat{v}', \widehat{\sigma}_3 \end{array}}{\widehat{\rho}, \widehat{\tau} \vdash e_1(e_2), \widehat{\sigma} \Downarrow \widehat{v}', \widehat{\sigma}_3} \begin{array}{l} \widehat{\varsigma} = \langle e_1(e_2), \widehat{\rho}, \widehat{\sigma}, \widehat{\tau} \rangle \\ \widehat{\ell} = \langle x, \widehat{\tau}' \rangle \\ \widehat{\tau}' = \widehat{next}(\widehat{\tau}, \widehat{\varsigma}) \end{array}$$

$$(\textsc{AIfT})\frac{\widehat{\rho}, \widehat{\tau} \vdash e_1, \widehat{\sigma} \Downarrow \widehat{v}_1, \widehat{\sigma}_1 \widehat{\rho}, \widehat{\tau} \vdash e_2, \widehat{\sigma}_1 \Downarrow \widehat{v}, \widehat{\sigma}_2}{\widehat{\rho}, \widehat{\tau} \vdash \texttt{if0}(e_1)\{e_2\}\{e_3\}, \widehat{\sigma} \Downarrow \widehat{v}, \widehat{\sigma}_2} 0 \in \lfloor \gamma \rfloor_0(\widehat{v}_1)$$

$$(\textsc{AIfF})\frac{\widehat{\rho}, \widehat{\tau} \vdash e_1, \widehat{\sigma} \Downarrow \widehat{v}_1, \widehat{\sigma}_1 \qquad \widehat{\rho}, \widehat{\tau} \vdash e_3, \widehat{\sigma}_1 \Downarrow \widehat{v}, \widehat{\sigma}_2}{\widehat{\rho}, \widehat{\tau} \vdash \texttt{if0}(e_1)\{e_2\}\{e_3\}, \widehat{\sigma} \Downarrow \widehat{v}, \widehat{\sigma}_2} \neg 0 \in \lfloor \gamma \rfloor_{\neg 0}(\widehat{v}_1)$$

Figure 6.15: Big-step Abstract Evaluation Semantics

(Abstract Reachability) $\boxed{\widehat{\rho}, \widehat{\tau} \vdash e, \widehat{\sigma} \Uparrow \widehat{\varsigma}}$

$$(\text{AREFL})\frac{}{\widehat{\rho}, \widehat{\tau} \vdash e, \widehat{\sigma} \Uparrow \langle e, \widehat{\rho}, \widehat{\sigma}, \widehat{\tau} \rangle} \qquad (\text{ARBIN1})\frac{\widehat{\rho}, \widehat{\tau} \vdash e_1, \widehat{\sigma} \Uparrow \varsigma}{\widehat{\rho}, \widehat{\tau} \vdash b(e_1, e_2), \widehat{\sigma} \Uparrow \widehat{\varsigma}}$$

$$(\text{ARBIN2})\frac{\widehat{\rho}, \widehat{\tau} \vdash e_1, \widehat{\sigma} \Downarrow \widehat{v}_1, \widehat{\sigma}_1 \qquad \widehat{\rho}, \widehat{\tau} \vdash e_2, \widehat{\sigma}_1 \Uparrow \widehat{\varsigma}}{\widehat{\rho}, \widehat{\tau} \vdash b(e_1, e_2), \widehat{\sigma} \Uparrow \widehat{\varsigma}}$$

$$(\text{ARAPP1})\frac{\widehat{\rho}, \widehat{\tau} \vdash e_1, \widehat{\sigma} \Uparrow \widehat{\varsigma}}{\widehat{\rho}, \widehat{\tau} \vdash e_1(e_2), \widehat{\sigma} \Uparrow \widehat{\varsigma}}$$

$$(\text{ARAPP2})\frac{\widehat{\rho}, \widehat{\tau} \vdash e_1, \widehat{\sigma} \Downarrow \widehat{v}_1, \widehat{\sigma}_1 \qquad \widehat{\rho}, \widehat{\tau} \vdash e_2, \widehat{\sigma}_1 \Uparrow \widehat{\varsigma}}{\widehat{\rho}, \widehat{\tau} \vdash e_1(e_2), \widehat{\sigma} \Uparrow \widehat{\varsigma}} \langle \lambda x.e', \widehat{\rho}' \rangle \in \lfloor \gamma \rfloor_{clo}(\widehat{v}_1)$$

$$(\text{ARAPP3})\frac{\begin{array}{c}\widehat{\rho}, \widehat{\tau} \vdash e_1, \widehat{\sigma} \Downarrow \widehat{v}_1, \widehat{\sigma}_1 \\ \widehat{\rho}, \widehat{\tau} \vdash e_2, \widehat{\sigma}_1 \Downarrow \widehat{v}_2, \widehat{\sigma}_2 \\ \widehat{\rho}'[x \mapsto \widehat{\ell}], \widehat{\tau}' \vdash e', \widehat{\sigma}_2 \sqcup [\widehat{\ell} \mapsto \widehat{v}_2] \Uparrow \widehat{\varsigma}\end{array}}{\widehat{\rho}, \widehat{\tau} \vdash e_1(e_2), \widehat{\sigma} \Uparrow \widehat{\varsigma}} \begin{array}{r}\langle \lambda x.e', \widehat{\rho}' \rangle \in \lfloor \gamma \rfloor_{clo}(\widehat{v}_1) \\ \widehat{\varsigma} = \langle e_1(e_2), \widehat{\rho}, \widehat{\sigma}, \widehat{\tau} \rangle \\ \widehat{\ell} = \langle x, \widehat{\tau}' \rangle \\ \widehat{\tau}' = \widehat{next}(\widehat{\tau}, \widehat{\varsigma})\end{array}$$

$$(\text{ARIF1})\frac{\widehat{\rho}, \widehat{\tau} \vdash e_1, \widehat{\sigma} \Uparrow \widehat{\varsigma}}{\widehat{\rho}, \widehat{\tau} \vdash \texttt{if0}(e_1)\{e_2\}\{e_3\}, \widehat{\sigma} \Uparrow \widehat{\varsigma}}$$

$$(\text{ARIFT})\frac{\widehat{\rho}, \widehat{\tau} \vdash e_1, \widehat{\sigma} \Downarrow \widehat{v}_1, \widehat{\sigma}_1 \qquad \widehat{\rho}, \widehat{\tau} \vdash e_2, \widehat{\sigma}_1 \Uparrow \widehat{\varsigma}}{\widehat{\rho}, \widehat{\tau} \vdash \texttt{if0}(e_1)\{e_2\}\{e_3\}, \widehat{\sigma} \Uparrow \widehat{\varsigma}} 0 \in \lfloor \gamma \rfloor_0(\widehat{v}_1)$$

$$(\text{ARIFF})\frac{\widehat{\rho}, \widehat{\tau} \vdash e_1, \widehat{\sigma} \Downarrow \widehat{v}_1, \widehat{\sigma}_1 \qquad \widehat{\rho}, \widehat{\tau} \vdash e_3, \widehat{\sigma}_1 \Uparrow \widehat{\varsigma}}{\widehat{\rho}, \widehat{\tau} \vdash \texttt{if0}(e_1)\{e_2\}\{e_3\}, \widehat{\sigma} \Uparrow \widehat{\varsigma}} \neg 0 \in \lfloor \gamma \rfloor_{\neg 0}(\widehat{v}_1)$$

Figure 6.16: Big-step Abstract Reachability Semantics

Theorem 9 (Abstract Reachability Semantics Soundness)**.**

$$\begin{array}{rl} \mathit{If} & \rho, \tau \vdash e, \widetilde{\sigma} \Uparrow \langle e', \rho', \widetilde{\sigma}', \tau' \rangle \quad \mathit{and} \quad \rho', \tau' \vdash e', \widetilde{\sigma}' \Downarrow \widetilde{v}, \widetilde{\sigma}'' \\ \mathit{where} & \eta(\rho) \sqsubseteq \widehat{\rho} \quad \mathit{and} \quad \eta(\tau) \sqsubseteq \widehat{\tau} \quad \mathit{and} \quad \eta(\widetilde{\sigma}) \sqsubseteq \widehat{\sigma} \\ \mathit{then} & \widehat{\rho}, \widehat{\tau} \vdash e, \widehat{\sigma} \Uparrow \langle e', \widehat{\rho}', \widehat{\sigma}', \widehat{\tau}' \rangle \quad \mathit{and} \quad \widehat{\rho}', \widehat{\tau}' \vdash e, \widehat{\sigma}' \Downarrow \widehat{v}, \widehat{\sigma}'' \\ \mathit{where} & \eta(\rho') \sqsubseteq \widehat{\rho}', \eta(\tau') \sqsubseteq \widehat{\tau}', \eta(\widetilde{\sigma}') \sqsubseteq \widehat{\sigma}', v \in \widetilde{v}, \eta(\sigma'') \sqsubseteq \widetilde{\sigma}'' \end{array}$$

The proof is by induction on the big-step derivation. The extraction function η is defined separately for environments (ρ), time (τ), collecting stores ($\widetilde{\sigma}$), values ($\widetilde{v}$) and configurations ($\widetilde{\varsigma}$). $\eta(\tau)$ and $\eta(\widetilde{v})$ are given with parameters $\widehat{\mathit{time}}$ and $\widehat{\mathit{val}}$. $\eta(\rho)$, $\eta(\widetilde{\sigma})$ and $\eta(\widetilde{\varsigma})$ are defined pointwise:

$$\eta(\rho)(x) \coloneqq \eta(\rho(x)) \qquad\qquad \eta(\widetilde{\sigma})(\widehat{\ell}) \coloneqq \bigsqcup_{\ell \in \gamma(\widehat{\ell})} \eta(\widetilde{\sigma}(\ell))$$

$$\eta(\langle e, \rho, \tau, \widetilde{\sigma} \rangle) \coloneqq \langle e, \eta(\rho), \eta(\tau), \eta(\widetilde{\sigma}) \rangle$$

Computing the Analysis An analysis for the program e_0 w.r.t. the abstract semantics is some cache $\$ \in \widehat{\mathit{config}} \mapsto \wp(\widehat{\mathit{val}} \times \widehat{\mathit{store}})$ that maps all configurations reachable from the initial configuration $\langle e_0, \widehat{\rho}_0, \widehat{\sigma}_0, \widehat{\tau}_0 \rangle$ to their final values and stores $\widehat{v}, \widehat{\sigma}$, which we notate $\$ \models e_0$:

$$\$ \models e_0 \quad \mathit{iff} \quad \begin{array}{rl} \mathit{If} & \widehat{\rho}_0, \widehat{\tau}_0 \vdash e_0, \widehat{\sigma}_0 \Uparrow \langle e, \widehat{\rho}, \widehat{\sigma}, \widehat{\tau} \rangle \\ \mathit{and} & \widehat{\rho}, \widehat{\tau} \vdash e, \widehat{\sigma} \Downarrow \widehat{v}, \widehat{\sigma}' \\ \mathit{then} & \langle \widehat{v}, \widehat{\sigma}' \rangle \in \$(\langle e, \widehat{\rho}, \widehat{\sigma}, \widehat{\tau} \rangle) \end{array}$$

The *best* cache $\$^+$ is then computed as the least fixed point of the functional $\mathcal{F}$:

$$\mathcal{F} \in (\widehat{\mathit{config}} \mapsto \wp(\widehat{\mathit{val}} \times \widehat{\mathit{store}})) \to (\widehat{\mathit{config}} \mapsto \wp(\widehat{\mathit{val}} \times \widehat{\mathit{store}}))$$

$$\mathcal{F} \coloneqq \lambda\$. \bigsqcup_{\langle e, \widehat{\rho}, \widehat{\sigma}, \widehat{\tau} \rangle \in \$} \begin{cases} \{\langle e, \widehat{\rho}, \widehat{\sigma}, \widehat{\tau} \rangle \mapsto \{\langle \widehat{v}, \widehat{\sigma}' \rangle\} \mid \widehat{\rho}, \widehat{\tau} \vdash e, \widehat{\sigma} \Downarrow^{\$} \widehat{v}, \widehat{\sigma}'\} \\ \{\widehat{\varsigma} \mapsto \{\} \mid \widehat{\rho}, \widehat{\tau} \vdash e, \widehat{\sigma} \Uparrow^{\$} \widehat{\varsigma}\} \end{cases}$$

which also includes the initial configuration:

$$\$^+ \coloneqq \mathit{lfp}(\lambda\$.\mathcal{F}(\$) \sqcup \{\langle e_0, \eta(\rho_0), \eta(\sigma_0), \eta(\tau_0)\rangle \mapsto \{\}\})$$

The relations $\widehat{\rho}, \widehat{\tau} \vdash e, \widehat{\sigma} \Downarrow^{\$} \widehat{v}, \widehat{\sigma}'$ and $\widehat{\rho}, \widehat{\tau} \vdash e, \widehat{\sigma} \Uparrow^{\$} \widehat{\varsigma}$ are modified versions of the original abstract semantics, but with recursive judgements replaced by $\langle \widehat{v}, \widehat{\sigma}' \rangle \in \$(e, \widehat{\rho}, \widehat{\sigma}, \widehat{\tau})$ and $\widehat{\varsigma} \in \$(e, \widehat{\rho}, \widehat{\sigma}, \widehat{\tau})$ respectively. Therefore $\mathcal{F}$ is not recursive; the recursion in the relations is lifted to the outer fixed-point of the analysis. Because the state space $\widehat{\mathit{config}} \mapsto \wp(\widehat{\mathit{val}} \times \widehat{\mathit{store}})$ is finite and $\mathcal{F}$ is monotonic, $\$^+$ can be computed algorithmically in finite time by Kleene fixed-point iteration. See Nielson et al. [1999] for more background and examples of static analyzers computed in this style, and from which the current development was largely inspired.

Theorem 10 (Algorithm Correctness)**.** $\$^+$ *is a valid analysis for* e_0*, that is:* $\$^+ \models e_0$.

The proof is by induction on the assumed derivations $\widehat{\rho}_0, \widehat{\tau}_0 \vdash e_0, \widehat{\sigma}_0 \Uparrow \langle \widehat{e}, \widehat{\rho}, \widehat{\sigma}, \widehat{\tau} \rangle$ and $\widehat{\rho}, \widehat{\tau} \vdash e, \widehat{\sigma} \Downarrow \widehat{v}, \widehat{\sigma}'$, and utilizes the fact that $\$^+$ is a fixed point, that is: $\mathcal{F}(\$^+) = \$^+$. Our final theorem relates the analysis cache $\$^+$ back to the concrete semantics of the initial program as a sound approximation:

Theorem 11 (Algorithm Soundness)**.**

$$\begin{array}{rl} \mathit{If} & \rho_0, \tau_0 \vdash e_0, \sigma_0 \Uparrow \langle e, \rho, \sigma, \tau \rangle \quad \mathit{and} \quad \rho, \tau \vdash e, \sigma \Downarrow v, \sigma' \\ \mathit{then} & \langle \widehat{v}, \widehat{\sigma}' \rangle \in \$^+(\langle e, \widehat{\rho}, \widehat{\sigma}, \widehat{\tau} \rangle) \\ \mathit{where} & \eta(\rho) \sqsubseteq \widehat{\rho}, \eta(\tau) \sqsubseteq \widehat{\tau}, \eta(\sigma) \sqsubseteq \widehat{\sigma}, \eta(v) \sqsubseteq \widehat{v}, \eta(\sigma') \sqsubseteq \widehat{\sigma}' \end{array}$$

The proof follows by composing Theorems 1-4.

Computing with Definitional Interpreters The algorithm described in Section 6.4 is a more efficient strategy for computing $\$^+$ using an extensible open-recursive definitional interpreter. This technique is general, and bridges the gap between the big-step abstract semantics formalized in this section and the definitional interpreters we wish to execute to obtain analyses.

An extensible open-recursive definitional interpreter for $\lambda\texttt{IF}$ (the small language formalized in this section) has domain:

$$\mathcal{E} \in \Sigma \to \Sigma \quad \textit{where} \quad \Sigma := \widehat{\textit{config}} \to \wp(\widehat{\textit{val}} \times \widehat{\textit{store}})$$

and is defined such that its denotational-fixed-point ($Y(\mathcal{E})$) recovers concrete interpretation when instantiated with the concrete state-space. For example, the recursive case for binary operator expressions is defined:

$$\begin{aligned}&\mathcal{E}(\mathcal{E}')(\langle b(e_1, e_2), \widehat{\rho}, \widehat{\sigma}, \widehat{\tau}\rangle) :=\\ &\quad \{\widehat{[\![b]\!]}(\widehat{v}_1, \widehat{v}_2) \mid \langle \widehat{v}_1, \widehat{\sigma}_1\rangle \in \mathcal{E}'(\langle e_1, \widehat{\rho}, \widehat{\sigma}, \widehat{\tau}\rangle) \wedge \langle \widehat{v}_2, \widehat{\sigma}_2\rangle \in \mathcal{E}'(\langle e_2, \widehat{\rho}, \widehat{\sigma}_1, \widehat{\tau}\rangle)\}\end{aligned}$$

The iteration strategy to analyze the program e_0 is then to run e_0 using $\mathcal{E}$, but intercepting recursive calls to:

1. Cache results for all intermediate configurations $\widehat{\varsigma}$; and
2. Cache seen states to prevent infinite loops.

(1) is required to fulfill the specification that $\$^+$ include results for all reachable configurations from e_0, and (2) is required to reach a fixed point of the analysis. To track this extra information we add functional state to the interpreter (which was

done through a monad transformer in Section 6.4) of type:

$$\widehat{cache} \coloneqq \widehat{config} \mapsto \wp(\widehat{val} \times \widehat{store})$$

such that the open-recursive evaluator has type:

$$\mathcal{E} \in \Sigma \to \Sigma \quad \textit{where} \quad \Sigma \coloneqq \widehat{config} \times \widehat{cache} \to \wp(\widehat{val} \times \widehat{store}) \times \widehat{cache}$$

The iteration to compute $\$^+$ given $\mathcal{E}$ is then defined:

$$\begin{aligned}
\$^+ \coloneqq\ & \mathit{lfp}(\lambda\$^o. \\
& \texttt{let}\ \mathcal{E}^* \coloneqq Y(\lambda\mathcal{E}'.\mathcal{E}(\lambda\langle\widehat{\varsigma}, \$^i\rangle. \\
& \qquad\qquad \texttt{if}\ \widehat{\varsigma} \in \$^i\ \texttt{then}\ \langle\$^i(\widehat{\varsigma}), \$^i\rangle\ \texttt{else} \\
& \qquad\qquad \texttt{let}\ \langle\widehat{VS}, \$^{i\prime}\rangle \coloneqq \mathcal{E}'(\widehat{\varsigma}, \$^i[\widehat{\varsigma} \mapsto \$^o(\widehat{\varsigma})]) \\
& \qquad\qquad \texttt{in}\ \langle\widehat{VS}, \$^{i\prime}[\widehat{\varsigma} \mapsto \widehat{VS}]\rangle))) \\
& \texttt{in}\ \pi_2(\mathcal{E}^*(\langle e_0, \widehat{\rho}_0, \widehat{\sigma}_0, \widehat{\tau}_0\rangle, \{\})))
\end{aligned}$$

The fixed interpreter $\mathcal{E}^*$ calls the unfixed interpreter $\mathcal{E}$, but intercepts recursive calls to perform (1) and (2) described above. When loops are detected, the results from the previous complete result $\$^o$ is used, and the outer fixed-point computes the least fixed point of this $\$^o$.

The end result is that, rather than compute analysis results and reachable states naively with Kleene fixed-point iteration, we are able to reuse the standard definitional interpreter—written in open-recursive form—to simultaneously explore reachable states, cache intermediate configurations, and iterate towards a least fixed-point solution for the analysis. This method is more efficient, and reuses an extensible definitional interpreter which can recover a wide range of analyses, including concrete interpretation.

WIDENING Two forms of widening can be employed to the semantics and iteration algorithm to achieve acceptable performance for the abstract interpreter.

The first form of widening is to widen the store in the result set $\wp(\widehat{val} \times \widehat{store})$ to $\wp(\widehat{val}) \times \widehat{store}$ in the evaluator $\mathcal{E}$:

$$\mathcal{E} \in \Sigma \to \Sigma \quad \textit{where} \quad \Sigma := \widehat{config} \times \widehat{cache} \to \wp(\widehat{val}) \times \widehat{store} \times \widehat{cache}$$

We perform this widening systematically and with no added effort through the use of Galois Transformers [Darais et al., 2015] in Section 6.6. The iteration strategy for this widened state space is the same as before, which computes a fixed point of the outer cache $\$^o$.

The next form of widening is to pull the store out of the configuration space *entirely*, that is:

$$\begin{aligned} \widehat{\varsigma} \in \widehat{config} &:= exp \times \widehat{env} \times \widehat{time} \\ \$ \in \widehat{cache} &:= \widehat{config} \mapsto \wp(\widehat{val}) \end{aligned}$$

and:

$$\mathcal{E} \in \Sigma \to \Sigma \quad \textit{where} \quad \Sigma := \widehat{config} \times \widehat{store} \times \widehat{cache} \to \wp(\widehat{val}) \times \widehat{store} \times \widehat{cache}$$

The fixed point iteration then finds a mutual least fixed-point of both the outer

cache $\$^o$ *and* the store $\widehat{\sigma}$:

$$
\begin{array}{l}
\langle \$^+, \widehat{\sigma}^+ \rangle \;\coloneqq\; \mathit{lfp}(\lambda \langle \$^o, \widehat{\sigma} \rangle . \\
\quad \texttt{let}\ \mathcal{E}^* \coloneqq Y(\lambda \mathcal{E}' . \mathcal{E}(\lambda \langle \widehat{\varsigma}, \widehat{\sigma}^i, \$^i \rangle . \\
\qquad\qquad\qquad \texttt{if}\ \widehat{\varsigma} \in \$^i\ \texttt{then}\ \langle \$^i(\widehat{\varsigma}), \sigma^i, \$^i \rangle\ \texttt{else} \\
\qquad\qquad\qquad \texttt{let}\ \langle \widehat{V}, \widehat{\sigma}^{i\prime}, \$^{i\prime} \rangle \coloneqq \mathcal{E}'(\widehat{\varsigma}, \widehat{\sigma}^i, \$^i[\widehat{\varsigma} \mapsto \$^o(\widehat{\varsigma})]) \\
\qquad\qquad\qquad \texttt{in}\ \langle \widehat{V}, \widehat{\sigma}^{i\prime}, \$^{i\prime}[\widehat{\varsigma} \mapsto \widehat{V}] \rangle))) \\
\quad \texttt{in}\ \pi_{2\times 3}(\mathcal{E}^*(\langle e_0, \widehat{\rho}_0, \widehat{\tau}_0 \rangle, \widehat{\sigma}, \{\})))
\end{array}
$$

This second version of widening, which computes a fixed-point also over the store, recovers a so-called *flow-insensitive* analysis. In this model, all program states are re-analyzed in the store resulting from execution. Also, the cache ($) does not index over store states $\widehat{\sigma}$ in its domain, greatly reducing its size, and leading to a much more efficient (although less precise) static analyzer.

Recovering Classical 0CFA From the fully widened static analyzer, which computes a mutual fixed-point between a cache and store, we can easily recover a classical 0CFA analysis. We do this by instantiating $\widehat{\mathit{time}}$ to the singleton abstraction $\{\bullet\}$, as was shown in Section 6.3. In this setting, the lexical environment ρ is uniquely determined by the program expression e, and can therefore be eliminated, resulting in the analysis state space:

$$
\begin{array}{rcl}
\widehat{\varsigma} \in \widehat{\mathit{config}} & \coloneqq & \mathit{exp} \\
\$ \in \widehat{\mathit{cache}} & \coloneqq & \mathit{exp} \mapsto \wp(\widehat{\mathit{val}}) \\
\widehat{\sigma} \in \widehat{\mathit{store}} & \coloneqq & \mathit{var} \mapsto \wp(\widehat{\mathit{val}})
\end{array}
$$

The specification for the analysis and the fully store-widened least fixed-point iteration for computing it recovers the constraint-based description of 0CFA given by Nielson

et al. [1999], where 0CFA is defined as the smallest cache ($) and store ($\sigma$) which satisfy a co-inductively defined judgment: $\$, \sigma \models e$.

Recovering Pushdown Analysis We borrow from the recent result in pushdown analysis by Gilray et al. [2016b] which shows that full pushdown precision can be achieved in a small-step store-widened abstract semantics by allocating continuations using a particular address space: program expressions paired with abstract environments ($\langle e, \widehat{\rho} \rangle$). In other words, $\langle e, \widehat{\rho} \rangle$ is sufficient to achieve full pushdown precision because the tuple uniquely identifies the evaluation context up to the final result of evaluation.

Our fully widened semantics recovers pushdown precision because the cache maps tuples $\langle e, \widehat{\rho}, \widehat{\tau} \rangle$, which contains $\langle e, \widehat{\rho} \rangle$. We then see that abstract time $\widehat{\tau}$ is redundant and eliminate it from the cache, resulting in a smaller domain for the same analysis:

$$\begin{aligned}
\widehat{\varsigma} \in \widehat{\mathit{config}} &::= \mathit{exp} \times \widehat{\mathit{env}} \times \widehat{\mathit{time}} \\
\$ \in \widehat{\mathit{cache}} &::= \mathit{exp} \times \widehat{\mathit{env}} \mapsto \wp(\widehat{\mathit{val}}) \\
\widehat{\sigma} \in \widehat{\mathit{store}} &::= \mathit{var} \times \widehat{\mathit{addr}} \mapsto \wp(\widehat{\mathit{val}})
\end{aligned}$$

An advantage of our setting is that we recover pushdown analysis also for varying degrees of store-widening, which is not the case in Gilray et al., although pushdown precision for non-widened semantics has been achieved by Johnson and Van Horn [Johnson and Van Horn, 2014]. Furthermore, the implementation of our analyzer inherits this precision through precise call-return matching in the defining metalanguage, requiring no added instrumentation to the state-space of the analyzer.

Going back to Nielson et al. [1999], it would be interesting to redevelop their constraint-based analysis descriptions of kCFA in a form that recovers pushdown precision. Such an exercise would amount to translating our big-step abstract semantics instantiated to kCFA to a constraint system. The resulting system would differ from classical kCFA by the addition of environments $\widehat{\rho}$ (which Nielson *et al.* call context environments) to the domain of the cache. In this way our formal framework is able to bridge the gap between results in pushdown analysis described *via* small-step machines *à la* Van Horn and Might [Van Horn and Might, 2010], and constraint-based systems *à la* Nielson *et al.* for which pushdown analysis has yet to be described effectively.

6.11 Related Work

This work draws upon and re-presents many ideas from the literature on abstract interpretation for higher-order languages [Midtgaard, 2012]. In particular, it closely follows the abstracting abstract machines [Van Horn and Might, 2010, 2012] approach to deriving abstract interpreters from a small-step machine. The key difference here is that we operate in the setting of a monadic definitional interpreter instead of an abstract machine. In moving to this new setting we developed a novel caching mechanism and fixed-point algorithm, but otherwise followed the same recipe. Remarkably, in the setting of definitional interpreters, the pushdown property for the analysis is simply inherited from the meta-language rather than requiring explicit instrumentation to the abstract interpreter.

Compositionally defined abstract interpretation functions for higher-order languages were first explored by Jones and Nielson [1995], which introduces the technique of interpreting a higher-order object language directly as terms in a meta-language to perform abstract interpretation. While their work lays the foundations for this idea, it does not consider abstractions for fixed-points in the domain, so although their abstract interpreters are sound, they are not in general computable. They propose a naïve solution of truncating the interpretation of syntactic fixed-points to some finite depth, but this solution isn't general and doesn't account for non-syntactic occurrences of bottom in the concrete domain (*e.g.*, *via* Y combinators). Our work develops such an abstraction for concrete denotational fixed-points using a fixed-point caching algorithm, resulting in general, computable abstractions for arbitrary definitional interpreters.

The use of monads and monad transformers to make extensible (concrete) interpreters is a well-known idea [Liang et al., 1995, Moggi, 1989, Steele, 1994], which we have extended to work for compositional abstract interpreters. The use of monads and monad transformers in machine-based formulations of abstract interpreters has previously been explored by Sergey et al. [2013] and Darais et al. [2015], respectively, and inspired our own adoption of these ideas. Darais has also shown that certain monad transformers are also *Galois transformers*, *i.e.*, they compose to form monads that transport Galois connections. Using Galois transformers may enable both compositional code *and proofs* for abstract interpreters in the style presented here.

The caching mechanism used to ensure termination in our abstract interpreter is similar to that used by Johnson and Van Horn [2014]. They use a local- and

meta-memoization table in a machine-based interpreter to ensure termination for a pushdown abstract interpreter. This mechanism is in turn reminiscent of Glück's use of memoization in an interpreter for two-way non-deterministic pushdown automata [Glück, 2013].

Caching recursive, non-deterministic functions is a well-studied problem in the functional logic programming community through a technique called "tabling" [Bol and Degerstedt, 1993, Chen and Warren, 1996, Swift and Warren, 2012, Tamaki and Sato, 1986], which has been successfully applied to program verification and analysis [Dawson et al., 1996, Janssens and Sagonas, 1998]. Unlike these systems, our approach uses a shallow embedding of cached non-determinism that can be applied in general-purpose functional languages. Monad transformers that enable shallow embedding of cached non-determinism are of continued interest since Hinze's *Deriving Backtracking Monad Transformers* [Fischer et al., 2009, Hinze, 2000, Kiselyov et al., 2005], and recent work [Ploeg and Kiselyov, 2014, Vandenbroucke et al., 2015] points to potential optimizations that can be applied to our naive iteration strategy.

Vardoulakis, who was the first to develop the idea of a pushdown abstraction for higher-order flow analysis [Vardoulakis and Shivers, 2011], formalized CFA2 using a CPS model, which is similar in spirit to a machine-based model. However, in his dissertation [Vardoulakis, 2012] he sketches an alternative presentation dubbed "Big CFA2" which is a big-step operational semantics for doing pushdown analysis quite similar in spirit to the approach presented here. One key difference is that Big CFA2 fixes a particular coarse abstraction of base values and closures—for example, both branches of a conditional are always evaluated. Consequently, it only uses

a single iteration of the abstract evaluation function, and avoids the need for the cache-based fixed-point of Section 6.4. We believe Big CFA2 as stated is sound, however if the underlying abstractions were tightened, it may then require a more involved fixed-point finding algorithm like the one we developed.

Our formulation of a pushdown abstract interpreter computes an abstraction similar to the many existing variants of pushdown flow analysis [Earl et al., 2010, Gilray et al., 2016b, Johnson and Van Horn, 2014, Van Horn and Might, 2012, Vardoulakis, 2012, Vardoulakis and Shivers, 2011]. Our incorporation of an abstract garbage collector into a pushdown abstract interpreter achieves a similar goal as that of so-called *introspective* pushdown abstract interpreters [Earl et al., 2012, Johnson et al., 2014]. The mixing of symbolic execution and abstract interpretation is similar in spirit to the *logic flow analysis* of Might [Might, 2007b], albeit in a pushdown setting and with a stronger notion of negation; generally, our presentation resembles traditional formulations of symbolic execution more closely [King, 1976]. Our approach to symbolic execution only handles the first-order case of symbolic values, as is common. However, Nguyễn's work on higher-order symbolic execution [Nguyễn and Van Horn, 2015] demonstrates how to scale to behavioral symbolic values. In principle, it should be possible to handle this case in our approach by adapting Nguyễn's method to a formulation in a compositional evaluator, but this remains to be carried out.

Now that we have abstract interpreters formulated with a basis in abstract machines and with a basis in monadic interpreters, an obvious question is can we obtain a correspondence between them similar to the functional correspondence

between their concrete counterparts [Ager et al., 2005]. An interesting direction for future work is to try to apply the usual tools of defunctionalization, CPS, and refocusing to see if we can interderive these abstract semantic artifacts.

6.12 Conclusions

We have shown that the AAM methodology can be adapted to definitional interpreters written in monadic style. Doing so captures a wide variety of semantics, such as the usual concrete semantics, collecting semantics, and various abstract interpretations. Beyond recreating existing techniques from the literature such as store-widening and abstract garbage collection, we can also design novel abstractions and capture disparate forms of program analysis such as symbolic execution. Further, our approach enables the novel combination of these techniques.

To our surprise, the definitional abstract interpreter we obtained implements a form of pushdown control flow abstraction in which calls and returns are always properly matched in the abstract semantics. True to the definitional style of Reynolds, the evaluator involves no explicit mechanics to achieve this property; it is simply inherited from the metalanguage.

We believe this formulation of abstract interpretation offers a promising new foundation towards re-usable components for the static analysis and verification of higher-order programs. Moreover, we believe the definitional abstract interpreter approach to be a fruitful new perspective on an old topic. We are left wondering: what else can be profitably inherited from the metalanguage of an abstract interpreter?

Chapter 7: Concluding Remarks

In this thesis we have aimed to lower the barrier to adopting *high assurance program analyzers* for use in creating *reliable software systems*. These barriers are the *feasibility* of mechanically verifying individual program analyzers, and the degree to which general purpose program analysis machinery supports *reuse*. Without feasibility and reuse, program analyzers will never make a meaningful impact on the quality of software produced by practitioners.

Our first contribution, *Constructive Galois Connections*, addresses feasibility by making it possible to mechanically verify a large class of correct-by-construction program analyzers which previous approaches were unable to verify. This was achieved by solving an open problem from the literature which was the primary barrier to achieving mechanized verification for this class of analyzers. Using Constructive Galois Connections, it is now possible to synthesize correct-by-construction program analyzers directly from programming language semantics, all while remaining embedded in a mechanized verification framework which supports immediate extraction of verified program analyzers from the results of synthesis.

Our second contribution, *Galois Transformers*, makes it possible to reuse *program analysis machinery* across different program analyzer implementations. This

was achieved by isolating a large class of analyzer design decisions using a novel interface for separating these concerns. Using Galois Transformers, it is now possible to design a single program analyzer—for, say, Java or C programming languages—and tune each of context, object, path, and flow sensitivity for the analyzer, all without needing to modify the implementation. The ability to tune these precision parameters is important for practitioners because there is no one-size-fits-all point in their design space. *E.g.*, analyzing buffer overflows requires a very different instantiation for these parameters than analyzing data integrity and confidentiality.

Our third and final contribution, *Abstracting Definitional Interpreters*, makes it possible to reuse *programming language features* across different program analyzer implementations. This was achieved by transplanting an existing systematic approach for designing program analyzers into a new setting which supports plug-and-play composition of programming language features. Using Abstracting Definitional Interpreters, it is now possible to design a single program analyzer—for, say, Ruby or Python—merely as the composition of its programming language features. The ability to quickly construct new analyzers from existing components is becoming more and more important as the number of programming languages used by practitioners continues to expand. *E.g.*, Ruby and Python share many language features in common (object-orientation, first-class procedures, late binding, etc.) and our work paves the way towards a modular analysis tool which supports a wide range of similar programming languages, as opposed to most tools which only support one language.

Appendix A: Galois Transformer Proofs

A.0.1 Lemma 5 [Galois Transformers] (Section 5.8.4)

STATE *$S^t[s]$ is a Galois transformer.*

Recall the definition of $S^t[s]$ and $\Pi^{S^t}[s]$:

$$\begin{aligned} S^t[s](m)(A) &\coloneqq s \to m(A \times s) \\ \Pi^{S^t}[s](\Sigma)(A) &\coloneqq \Sigma(A \times s) \to \Sigma(A \times s) \end{aligned}$$

STATE PROPERTY (1) The action $S^t[s]$ on functions:

$$\begin{aligned} S^t[s] &: (A \to m(B)) \to A \to S^t[s](m)(B) \\ S^t[s](f)(x)(s) &\coloneqq y \leftarrow^m f(x)\,;\ \mathit{return}^m(y, s) \end{aligned}$$

To transport Galois connections, we assume a Galois connection $A \to m_1(B) \underset{\alpha^m}{\overset{\gamma^m}{\leftrightarrows}} A \to m_2(B)$ and define α and γ:

$$\begin{aligned} \alpha &: (A \to S^t[s](m_1)(B)) \to A \to S^t[s](m_2)(B) \\ \gamma &: (A \to S^t[s](m_2)(B)) \to A \to S^t[s](m_1)(B) \end{aligned}$$

$$\begin{aligned} \alpha(f)(x)(s) &\coloneqq \alpha^m(\lambda\langle x, s\rangle.f(x)(s))(x, s) \\ \gamma(f)(x)(s) &\coloneqq \gamma^m(\lambda\langle x, s\rangle.f(x)(s))(x, s) \end{aligned}$$

α and γ are monotonic by inspection, and extensive and reductive:

$$\textit{extensive} \ : \ \forall f x s. f(x)(s) \sqsubseteq \gamma(\alpha(f))(x)(s)$$

$$\gamma(\alpha(f))(x)(s)$$

$$= \ \wr \ \text{definition of } \alpha \text{ and } \gamma \ \int$$

$$\gamma^m(\lambda\langle x, s\rangle.\alpha^m(\lambda\langle x, s\rangle.f(x)(s))(x, s))(x, s)$$

$$= \ \wr \ \eta\text{-reduction} \ \int$$

$$\gamma^m(\alpha^m(\lambda\langle x, s\rangle.f(x)(s)))(x, s)$$

$$\sqsupseteq \ \wr \ \gamma^m \circ \alpha^m \text{ extensive} \ \int$$

$$(\lambda\langle x, s\rangle.f(x)(s))(x, s)$$

$$= \ \wr \ \beta\text{-reduction} \ \int$$

$$f(x)(s) \quad \blacksquare$$

$$\textit{reductive} \ : \ \forall f x s. \alpha(\gamma(f))(x)(s) \sqsubseteq f(x)(s)$$

$$\alpha(\gamma(f))(x)(s)$$

$$= \ \wr \ \text{definition of } \alpha \text{ and } \gamma \ \int$$

$$\alpha^m(\lambda\langle x, s\rangle.\gamma^m(\lambda\langle x, s\rangle.f(x)(s))(x, s))(x, s)$$

$$= \ \wr \ \eta\text{-reduction} \ \int$$

$$\alpha^m(\gamma^m(\lambda\langle x, s\rangle.f(x)(s))(x, s))(x, s)$$

$$\sqsubseteq \ \wr \ \alpha^m \circ \gamma^m \text{ reductive} \ \int$$

$$(\lambda\langle x, s\rangle.f(x)(s))(x, s)$$

$$= \quad \wr \quad \beta\text{-reduction} \quad \int$$

$$f(x)(s) \quad \blacksquare$$

Finally, Property (1) commutes, assuming that $A \to m_1(B) \underset{\alpha^m}{\overset{\gamma^m}{\leftrightarrows}} A \to m_2(B)$ is homomorphic:

$$\mathit{goal} \; : \; S^t[s][m_2](\alpha^m(f))(x)(s) = \alpha(S^t[s][m_1](f))(x)(s)$$

$$\alpha(S^t[s][m_1](f))(x)(s)$$

$$= \quad \wr \quad \text{definition of } \alpha \text{ and } S^t[s][m_1] \quad \int$$

$$\alpha^m(\lambda\langle x, s\rangle.y \leftarrow^{m_1} f(x) \; ; \; \mathit{return}^{m_1}(y, s))(s, x)$$

$$= \quad \wr \quad \alpha^m \text{ homomorphic on } \mathit{bind}^{m_1} \text{ and } \mathit{return}^{m_1} \quad \int$$

$$(\lambda\langle x, s\rangle.y \leftarrow^{m_1} \alpha^m(f)(x) \; ; \; \mathit{return}^{m_2}(y, s))(s, x)$$

$$= \quad \wr \quad \beta\text{-reduction} \quad \int$$

$$y \leftarrow^{m_2} \alpha^m(f)(x) \; ; \; \mathit{return}^{m_2}(y, s)$$

$$= \quad \wr \quad \text{definition of } S^t[s] \quad \int$$

$$S^t[s][m_2](\alpha^m(f))(s)(x) \quad \blacksquare$$

State Property (2) The action $\Pi^{S^t}[s]$ on functions uses the mapping to monadic functions defined in Property (3):

$$\Pi^{S^t}[s] \quad : \quad (\Sigma(A) \to \Sigma(B)) \to \Pi^{S^t}[s](\Sigma)(A) \to \Pi^{S^t}[s](\Sigma)(B)$$
$$\Pi^{S^t}[s](f)(\varsigma) \; \coloneqq \; \gamma^{\Sigma \leftrightarrow m}(S^t[s](\alpha^{\Sigma \leftrightarrow m}(f)))(\varsigma)$$

To transport Galois connections, we assume $\Sigma_1(A) \to \Sigma_1(B) \underset{\alpha^\Sigma}{\overset{\gamma^\Sigma}{\leftrightarrows}} \Sigma_2(A) \to \Sigma_2(B)$ and define α and γ as instantiations of α^Σ and γ^Σ:

$$\begin{aligned}
\alpha &: (\Pi^{S^t}[s](\Sigma_1)(A) \to \Pi^{S^t}[s](\Sigma_1)(B)) \to \Pi^{S^t}[s](\Sigma_2)(A) \to \Pi^{S^t}[s](\Sigma_2)(B) \\
\gamma &: (\Pi^{S^t}[s](\Sigma_2)(A) \to \Pi^{S^t}[s](\Sigma_2)(B)) \to \Pi^{S^t}[s](\Sigma_1)(A) \to \Pi^{S^t}[s](\Sigma_1)(B)
\end{aligned}$$

$$\begin{aligned}
\gamma(f)(\varsigma) &\coloneqq \gamma^\Sigma(f)(\varsigma) \\
\alpha(f)(\varsigma) &\coloneqq \alpha^\Sigma(f)(\varsigma)
\end{aligned}$$

Monotonicity, reductive and extensive properties carry over by definition. Finally, Property (2) commutes, assuming that α^Σ and α^m commute with both $\gamma^{\Sigma\leftrightarrow m}$ and $\alpha^{\Sigma\leftrightarrow m}$:

$$\begin{aligned}
&\mathit{goal} : \Pi^{S^t}[s][\Sigma_2](\alpha^\Sigma(f))(\varsigma) = \alpha^\Sigma(\Pi^{S^t}[s][\Sigma_1](f))(\varsigma) \\
&\alpha^\Sigma(\Pi^{S^t}[s][\Sigma_1](f)(\varsigma) \\
&= \quad \wr \quad \text{definition of } \Pi^{S^t}[s][\Sigma_1] \quad \int \\
&\alpha^\Sigma(\gamma^{\Sigma\leftrightarrow m}(S^t[s](\alpha^{\Sigma\leftrightarrow m}(f))))(\varsigma) \\
&= \quad \wr \quad \text{definition of } S^t[s] \quad \int \\
&\alpha^\Sigma(\gamma^{\Sigma\leftrightarrow m}(\lambda x.\lambda s.y \leftarrow^{m_1} \alpha^{\Sigma\leftrightarrow m}(f)(x)\ ;\ \mathit{return}^{m_1}(y,s)))(\varsigma) \\
&= \quad \wr \quad \alpha^\Sigma \text{ and } \gamma^{\Sigma\leftrightarrow m} \text{ commute} \quad \int \\
&\gamma^{\Sigma\leftrightarrow m}(\alpha^m(\lambda x.\lambda s.y \leftarrow^{m_1} \alpha^{\Sigma\leftrightarrow m}(f)(x)\ ;\ \mathit{return}^{m_1}(y,s)))(\varsigma) \\
&= \quad \wr \quad \alpha^m \text{ homomorphic} \quad \int \\
&\gamma^{\Sigma\leftrightarrow m}(\lambda x.\lambda s.y \leftarrow^{m_2} \alpha^m(\alpha^{\Sigma\leftrightarrow m}(f))(x)\ ;\ \mathit{return}^{m_2}(y,s))(\varsigma) \\
&= \quad \wr \quad \alpha^m \text{ and } \alpha^{\Sigma\leftrightarrow m} \text{ commute} \quad \int \\
&\gamma^{\Sigma\leftrightarrow m}(\lambda x.\lambda s.y \leftarrow^{m_2} \alpha^{\Sigma\leftrightarrow m}(\alpha^\Sigma(f))(x)\ ;\ \mathit{return}^{m_2}(y,s))(\varsigma)
\end{aligned}$$

$= \quad \wr \quad$ definition of $S^t[s] \quad \int$

$\gamma^{\Sigma \leftrightarrow m}(S^t[s](\alpha^{\Sigma \leftrightarrow m}(\alpha^{\Sigma}(f))))(\varsigma)$

$= \quad \wr \quad$ definition of $\Pi^{S^t}[s][\Sigma_2] \quad \int$

$\Pi^{S^t}[s][\Sigma_2](\alpha^{\Sigma}(f))(\varsigma) \quad \blacksquare$

STATE PROPERTY (3) Assume a Galois connection $\Sigma(A) \to \Sigma(B) \underset{\alpha^{\Sigma \leftrightarrow m}}{\overset{\gamma^{\Sigma \leftrightarrow m}}{\leftrightarrows}} A \to m(B)$. The Galois connection between $S^t[s](m)$ and $\Pi^{S^t}[s](\Sigma)$ is defined:

$$\alpha \ : \ (\Pi^{S^t}[s](\Sigma)(A) \to \Pi^{S^t}[s](\Sigma)(B)) \to A \to S^t[s](m)(B)$$
$$\gamma \ : \ (A \to S^t[s](m)(B)) \to \Pi^{S^t}[s](\Sigma)(A) \to \Pi^{S^t}[s](\Sigma)(B)$$

$$\alpha(f)(x)(s) \ \coloneqq \ \alpha^{\Sigma \leftrightarrow m}(f)(x, s)$$
$$\gamma(f)(\varsigma) \ \coloneqq \ \gamma^{\Sigma \leftrightarrow m}(\lambda \langle x, s \rangle \to f(x)(s))(\varsigma)$$

α and γ are monotonic by inspection, and extensive and reductive:

$$\textit{extensive} \ : \ \forall f \varsigma . f(\varsigma) \sqsubseteq \gamma(\alpha(f))(\varsigma)$$

$\gamma(\alpha(f))(\varsigma)$

$= \quad \wr \quad$ definition of α and $\gamma \quad \int$

$\gamma^{\Sigma \leftrightarrow m}(\lambda \langle x, s \rangle \to \alpha^{\Sigma \leftrightarrow m}(f)(x, s))(\varsigma)$

$= \quad \wr \quad \eta$-reduction $\quad \int$

$\gamma^{\Sigma \leftrightarrow m}(\alpha^{\Sigma \leftrightarrow m}(f))(\varsigma)$

$\sqsupseteq \quad \wr \quad \gamma^{\Sigma \leftrightarrow m} \circ \alpha^{\Sigma \leftrightarrow m}$ extensive $\quad \int$

$f(\varsigma) \quad \blacksquare$

$$\textit{reductive} : \forall f x s.\alpha(\gamma(f))(x)(s) \sqsubseteq f(x)(s)$$

$$\begin{array}{ll} & \alpha(\gamma(f))(x)(s) \\ = & \wr \quad \text{definition of } \alpha \text{ and } \gamma \quad \int \\ & \alpha^{\Sigma\leftrightarrow m}(\gamma^{\Sigma\leftrightarrow m}(\lambda\langle x,s\rangle \to f(x)(s)))(x,s) \\ \sqsubseteq & \wr \quad \alpha^{\Sigma\leftrightarrow m} \circ \gamma^{\Sigma\leftrightarrow m} \text{ reductive} \quad \int \\ & (\lambda\langle x,s\rangle \to f(x)(s))(x,s) \\ = & \wr \quad \beta\text{-reduction} \quad \int \\ & f(x)(s) \quad \blacksquare \end{array}$$

Finally, Property (3) commutes:

$$\textit{goal} : \Pi^{S^t}[s][\Sigma](\gamma^{\Sigma\leftrightarrow m}(f))(\varsigma) \sqsubseteq \gamma(S^t[s](f))(\varsigma)$$

$$\begin{array}{ll} & \Pi^{S^t}[s][\Sigma](\gamma^{\Sigma\leftrightarrow m}(f))(\varsigma) \\ = & \wr \quad \text{definition of } \Pi^{S^t}[s][\Sigma] \quad \int \\ & \gamma^{\Sigma\leftrightarrow m}(\lambda\langle x,s\rangle \to S^t[s](\alpha^{\Sigma\leftrightarrow m}(\gamma^{\Sigma\leftrightarrow m}(f)))(x)(s))(\varsigma) \\ \sqsubseteq & \wr \quad \alpha^{\Sigma\leftrightarrow m} \circ \gamma^{\Sigma\leftrightarrow m} \text{ reductive} \quad \int \\ & \gamma^{\Sigma\leftrightarrow m}(\lambda\langle x,s\rangle \to S^t[s](f)(x)(s))(\varsigma) \\ = & \wr \quad \text{definition of } \gamma \quad \int \\ & \gamma(S^t[s](f))(\varsigma) \quad \blacksquare \end{array}$$

NONDETERMINISM *$\wp^t$ is a Galois transformer.*

Recall the definition of $\wp^t$ and $\Pi^{\wp^t}$:

$$\wp^t(m)(A) \coloneqq m(\wp(A)) \qquad\qquad \Pi^{\wp^t}(\Sigma)(A) \coloneqq \Sigma(\wp(A))$$

NONDETERMINISM PROPERTY (1) The action $\wp^t$ on functions:

$$\begin{aligned} \wp^t &: (A \to m(B)) \to A \to \wp^t(m)(B) \\ \wp^t(f)(x) &\coloneqq y \leftarrow^m f(x)\,;\ \mathit{return}^m(y) \end{aligned}$$

To transport Galois connections, we assume a Galois connection $A \to m_1(B) \underset{\alpha^m}{\overset{\gamma^m}{\leftrightarrows}} A \to m_2(B)$ define α and γ:

$$\begin{aligned} \alpha &: (A \to \wp(m_1)(B)) \to A \to \wp(m_2)(B) \\ \gamma &: (A \to \wp(m_2)(B)) \to A \to \wp(m_1)(B) \end{aligned}$$

$$\begin{aligned} \alpha(f)(x) &\coloneqq \alpha^m(\lambda\{x_1,\ldots,x_n\}.f(x_1) \sqcup^{m_1} \cdots \sqcup^{m_1} f(x_n))(\{x\}) \\ \gamma(f)(x) &\coloneqq \gamma^m(\lambda\{x_1,\ldots,x_n\}.f(x_1) \sqcup^{m_2} \cdots \sqcup^{m_2} f(x_n))(\{x\}) \end{aligned}$$

α and γ are monotonic by inspection, and extensive and reductive:

$\mathit{extensive} : \forall fx.f(x) \sqsubseteq \gamma(\alpha(f))(x)$

$\gamma(\alpha(f))(x)$

$= \ \wr$ definition of α and γ $\int$

$\gamma^m(\lambda\{x_1,\ldots,x_n\}.$

$\alpha^m(\lambda\{x_1,\ldots,x_n\}.f(x_1) \sqcup^{m_1} \cdots \sqcup^{m_1} f(x_n))(\{x_1\})$

$\sqcup^{m_2} \ldots \sqcup^{m_2}$

$\alpha^m(\lambda\{x_1,\ldots,x_n\}.f(x_1) \sqcup^{m_1} \cdots \sqcup^{m_1} f(x_n))(\{x_n\}))(\{x\})$

$= \quad \wr \quad$ left-unit of m_2 $\quad \int$

$\gamma^m(\lambda\{x_1, \ldots, x_n\}.$

$(\{x_1, \ldots, x_n\} \leftarrow^{m_2} \mathit{return}^{m_2}(\{x_1\})\ ;\ \alpha^m(\lambda\{x_1, \ldots, x_n\}.$

$f(x_1) \sqcup^{m_1} \cdots \sqcup^{m_1} f(x_n))(\{x_1, \ldots, x_n\}))$

$\sqcup^{m_2} \ldots \sqcup^{m_2}$

$(\{x_1, \ldots, x_n\} \leftarrow^{m_2} \mathit{return}^{m_2}(\{x_n\})\ ;\ \alpha^m(\lambda\{x_1, \ldots, x_n\}.$

$f(x_1) \sqcup^{m_1} \cdots \sqcup^{m_1} f(x_n))(\{x_1, \ldots, x_n\})))(\{x\})$

$\sqsupseteq \quad \wr \quad \alpha^m \circ \gamma^m$ reductive $\quad \int$

$\gamma^m(\lambda\{x_1, \ldots, x_n\}.$

$(\{x_1, \ldots, x_n\} \leftarrow^{m_2} \alpha^m(\gamma^m(\mathit{return}^{m_2}(\{x_1\})))\ ;$

$\alpha^m(\lambda\{x_1, \ldots, x_n\}.f(x_1) \sqcup^{m_1} \cdots \sqcup^{m_1} f(x_n))(\{x_1, \ldots, x_n\}))$

$\sqcup^{m_2} \ldots \sqcup^{m_2}$

$(\{x_1, \ldots, x_n\} \leftarrow^{m_2} \alpha^m(\gamma^m(\mathit{return}^{m_2}(\{x_n\})))\ ;$

$\alpha^m(\lambda\{x_1, \ldots, x_n\}.f(x_1) \sqcup^{m_1} \cdots \sqcup^{m_1} f(x_n))(\{x_1, \ldots, x_n\})))(\{x\})$

$= \quad \wr \quad \alpha^m$ and γ^m homomorphic on bind^{m_2} and $\mathit{return}^{m_2} \quad \int$

$\gamma^m(\lambda\{x_1, \ldots, x_n\}.$

$(\alpha^m(\{x_1, \ldots, x_n\} \leftarrow^{m_1} \mathit{return}^{m_1}(\{x_1\})\ ;$

$f(x_1) \sqcup^{m_1} \cdots \sqcup^{m_1} f(x_n))) \sqcup^{m_2} \cdots \sqcup^{m_2}$

$(\alpha^m(\{x_1, \ldots, x_n\} \leftarrow^{m_1} \mathit{return}^{m_1}(\{x_n\})\ ;$

$f(x_1) \sqcup^{m_1} \cdots \sqcup^{m_1} f(x_n))))(\{x\})$

$= \quad \wr \quad$ join-semilattice functorality of $m \quad \int$

$\gamma^m(\alpha^m(\lambda\{x_1, \ldots, x_n\}.\{x_1, \ldots, x_n\} \leftarrow \mathit{return}^{m_1}(\{x_1, \ldots, x_n\})\,;$

$\quad f(x_1) \sqcup^{m_1} \cdots \sqcup^{m_1} f(x_n)))(\{x\})$

$\sqsupseteq \quad \wr \quad \gamma^m \circ \alpha^m$ extensive $\quad \int$

$\{x_1, \ldots, x_n\} \leftarrow \mathit{return}^{m_1}(\{x\})\,;\; f(x_1) \sqcup^{m_1} \cdots \sqcup^{m_1} f(x_n)$

$= \quad \wr \quad$ left-unit of $m \quad \int$

$f(x) \quad \blacksquare$

reductive : $\forall f x.\alpha(\gamma(f))(x) \sqsubseteq f(x)$

$\alpha(\gamma(f))(x)$

$= \quad \wr \quad$ definition of α and $\gamma \quad \int$

$\alpha^m(\lambda\{x_1, \ldots, x_n\}.$

$\quad \gamma^m(\lambda\{x_1, \ldots, x_n\}.f(x_1) \sqcup^{m_2} \cdots \sqcup^{m_2} f(x_n))(\{x_1\})$

$\quad \sqcup^{m_1} \cdots \sqcup^{m_1}$

$\quad \gamma^m(\lambda\{x_1, \ldots, x_n\}.f(x_1) \sqcup^{m_2} \cdots \sqcup^{m_2} f(x_n))(\{x_n\}))(\{x\})$

$= \quad \wr \quad$ left-unit of m_1 $\quad \int$

$\alpha^m(\lambda\{x_1, \ldots, x_n\}.$

$(\{x_1, \ldots, x_n\} \leftarrow^{m_1} \mathit{return}^{m_1}(\{x_1\})\ ;\ \gamma^m(\lambda\{x_1, \ldots, x_n\}.$

$f(x_1) \sqcup^{m_2} \cdots \sqcup^{m_2} f(x_n))(\{x_1, \ldots, x_n\}))$

$\sqcup^{m_1} \ldots \sqcup^{m_1}$

$(\{x_1, \ldots, x_n\} \leftarrow^{m_1} \mathit{return}^{m_1}(\{x_2\})\ ;\ \gamma^m(\lambda\{x_1, \ldots, x_n\}.$

$f(x_1) \sqcup^{m_2} \cdots \sqcup^{m_2} f(x_n))(\{x_1, \ldots, x_n\})))(\{x\})$

$\sqsubseteq \quad \wr \quad \gamma^m \circ \alpha^m$ extensive $\quad \int$

$\alpha^m(\lambda\{x_1, \ldots, x_n\}.$

$(\{x_1, \ldots, x_n\} \leftarrow^{m_1} \gamma^m(\alpha^m(\mathit{return}^{m_1}(\{x_1\})))\ ;$

$\gamma^m(\lambda\{x_1, \ldots, x_n\}.f(x_1) \sqcup^{m_2} \cdots \sqcup^{m_2} f(x_n))(\{x_1, \ldots, x_n\}))$

$\sqcup^{m_1} \ldots \sqcup^{m_1}$

$(\{x_1, \ldots, x_n\} \leftarrow^{m_1} \gamma^m(\alpha^m(\mathit{return}^{m_1}(\{x_n\})))\ ;$

$\gamma^m(\lambda\{x_1, \ldots, x_n\}.f(x_1) \sqcup^{m_2} \cdots \sqcup^{m_2} f(x_n))(\{x_1, \ldots, x_n\})))(\{x\})$

$= \quad \wr \quad \alpha^m \mathit{and} \gamma^m$ homomorphic on bind^{m_1} and $\mathit{return}^{m_1} \quad \int$

$\alpha^m(\lambda\{x_1, \ldots, x_n\}.$

$\gamma^m(\{x_1, \ldots, x_n\} \leftarrow^{m_2} \mathit{return}^{m_2}(\{x_1\})\ ;\ f(x_1) \sqcup^{m_2} \cdots \sqcup^{m_2} f(x_n))$

$\sqcup^{m_1} \ldots \sqcup^{m_1}$

$\gamma^m(\{x_1, \ldots, x_n\} \leftarrow^{m_2} \mathit{return}^{m_2}(\{x_1\})\ ;\ f(x_1) \sqcup^{m_2} \cdots \sqcup^{m_2} f(x_n)))(\{x\})$

$= \quad \wr$ join-semilattice functorailty of m $\int$

$\alpha^m(\gamma^m(\lambda\{x_1,\ldots,x_n\}.\{x_1,\ldots,x_n\} \leftarrow^{m_2} \mathit{return}^{m_2}(\{x_1,\ldots,x_n\})\,;$

$\quad f(x_1) \sqcup^{m_2} \cdots \sqcup^{m_2} f(x_n)))(\{x\})$

$\sqsubseteq \quad \wr$ $\alpha^m \circ \gamma^m$ reductive $\int$

$\{x_1,\ldots,x_n\} \leftarrow^{m_2} \mathit{return}^{m_2}(\{x\})\,;\ f(x_1) \sqcup^{m_2} \cdots \sqcup^{m_2} f(x_n)$

$= \quad \wr$ left-unit of m $\int$

$f(x) \quad \blacksquare$

Finally, Property (1) commutes, assuming that $A \to m_1(B) \underset{\alpha^m}{\overset{\gamma^m}{\leftrightarrows}} A \to m_2(B)$ is homomorphic:

$$\mathit{goal} : \forall fs.\wp^t[m_2](\alpha^m(f))(x) = \alpha(\wp^t[m_1](f))(x)$$

$\alpha(\wp^t[m_1](f))(x)$

$= \quad \wr$ definition of α and $\wp^t[m_1](f)$ $\int$

$\alpha^m(\lambda\{x_1,\ldots,x_n\}.$

$\quad (y \leftarrow^{m_1} f(x_1)\,;\ \mathit{return}^{m_1}(\{y\}))$

$\quad \sqcup^{m_1} \cdots \sqcup^{m_1}$

$\quad (y \leftarrow^{m_1} f(x_n)\,;\ \mathit{return}^{m_1}(\{y\})))(\{x\})$

$= \quad \wr$ homomorphic on bind^{m_1} and return^{m_1} $\int$

$y \leftarrow^{m_2} \alpha^m(f)(x)\,;\ \mathit{return}^{m_2}(\{y\})$

$$= \quad \wr \quad \text{definition of } \wp^t[m_2] \quad \int$$

$$\wp^t[m_2](\alpha^m(f))(x) \quad \blacksquare$$

NONDETERMINISM PROPERTY (2) The action $\Pi^{\wp^t}$ on functions uses the mapping to monadic functions defined in Property (3):

$$\begin{aligned} \Pi^{\wp^t} &: (\Sigma(A) \to \Sigma(B)) \to \Pi^{\wp^t}(\Sigma)(A) \to \Pi^{\wp^t}(\Sigma)(B) \\ \Pi^{\wp^t}(f)(\varsigma) &\coloneqq \gamma^{\Sigma \leftrightarrow \gamma}(\wp^t(\alpha^{\Sigma \leftrightarrow \gamma}(f))) \end{aligned}$$

To transport Galois connections, we assume $\Sigma_1(A) \to \Sigma_1(B) \underset{\alpha^\Sigma}{\overset{\gamma^\Sigma}{\leftrightarrows}} \Sigma_2(A) \to \Sigma_2(B)$ and define α and γ as instantiations of α^Σ and γ^Σ:

$$\begin{aligned} \alpha &: (\Pi^{\wp^t}(\Sigma_1)(A) \to \Pi^{\wp^t}(\Sigma_1)(B)) \to \Pi^{\wp^t}(\Sigma_2)(A) \to \Pi^{\wp^t}(\Sigma_2)(B) \\ \gamma &: (\Pi^{\wp^t}(\Sigma_2)(A) \to \Pi^{\wp^t}(\Sigma_2)(B)) \to \Pi^{\wp^t}(\Sigma_1)(A) \to \Pi^{\wp^t}(\Sigma_1)(B) \end{aligned}$$

$$\begin{aligned} \alpha(f)(\varsigma) &\coloneqq \alpha^\Sigma(f)(\varsigma) \\ \gamma(f)(\varsigma) &\coloneqq \gamma^\Sigma(f)(\varsigma) \end{aligned}$$

Monotonicity, reductive and extensive properties carry over by definition. Finally, Property (2) commutes, assuming that α^Σ and α^m commute with both $\gamma^{\Sigma \leftrightarrow m}$ and $\alpha^{\Sigma \leftrightarrow m}$:

$$goal : \Pi^{\wp^t}[\Sigma_2](\alpha^\Sigma(f))(\varsigma) = \alpha^\Sigma(\Pi^{\wp^t}[\Sigma_1](f))(\varsigma)$$

$$\alpha^\Sigma(\Pi^{\wp^t}[\Sigma_1](f))(\varsigma)$$

$$= \quad \wr \quad \text{definition of } \Pi^{\wp^t} \quad \int$$

$$\alpha^\Sigma(\gamma^{\Sigma \leftrightarrow \gamma}(\wp^t(\alpha^{\Sigma \leftrightarrow \gamma}(f))))(\varsigma)$$

$$= \quad \wr \quad \text{definition of } \wp^t \quad \int$$

$$\alpha^\Sigma(\gamma^{\Sigma \leftrightarrow \gamma}(\lambda x.y \leftarrow^{m_1} \alpha^{\Sigma \leftrightarrow \gamma}(f)(x)\ ;\ return^{m_1}(\{y\})))(\varsigma)$$

$$= \quad \wr \quad \alpha^{\Sigma} \text{ and } \gamma^{\Sigma \leftrightarrow \gamma} \text{ commute} \quad \int$$

$$\gamma^{\Sigma \leftrightarrow \gamma}(\alpha^m(\lambda x.y \leftarrow^{m_1} \alpha^{\Sigma \leftrightarrow \gamma}(f)(x)\ ;\ \mathit{return}^{m_1}(\{y\})))(\varsigma)$$

$$= \quad \wr \quad \alpha^m \text{ homomorphic on } \mathit{bind}^{m_1} \text{ and } \mathit{return}^{m_2} \quad \int$$

$$\gamma^{\Sigma \leftrightarrow \gamma}(\lambda x.y \leftarrow^{m_2} \alpha^m(\alpha^{\Sigma \leftrightarrow \gamma}(f))(x)\ ;\ \mathit{return}^{m_2}(\{y\}))(\varsigma)$$

$$= \quad \wr \quad \alpha^m \text{ and } \alpha^{\Sigma \leftrightarrow \gamma} \text{ commute} \quad \int$$

$$\gamma^{\Sigma \leftrightarrow \gamma}(\lambda x.y \leftarrow^{m_2} \alpha^{\Sigma \leftrightarrow \gamma}(\alpha^{\Sigma}(f))(x)\ ;\ \mathit{return}^{m_2}(\{y\}))(\varsigma)$$

$$= \quad \wr \quad \text{definition of } \Pi^{\wp^t}[\Sigma_2] \text{ and } \alpha^{\Sigma} \quad \int$$

$$\Pi^{\wp^t}[\Sigma_2](\alpha^{\Sigma}(f))(\varsigma) \quad \blacksquare$$

Nondeterminism Property (3) Assume a Galois connection $\Sigma(A) \to \Sigma(B) \underset{\alpha^{\Sigma \leftrightarrow m}}{\overset{\gamma^{\Sigma \leftrightarrow m}}{\leftrightarrows}} A \to m(B)$. The Galois connection between $\wp^t(m)$ and $\Pi^{\wp^t}(\Sigma)$ is:

$$\begin{aligned} \alpha &: (\Pi^{\wp^t}(\Sigma)(A) \to \Pi^{\wp^t}(\Sigma)(B)) \to A \to \wp^t(m)(B) \\ \gamma &: (A \to \wp^t(m)(B)) \to \Pi^{\wp^t}(\Sigma)(A) \to \Pi^{\wp^t}(\Sigma)(B) \end{aligned}$$

$$\begin{aligned} \alpha(f)(x) &\coloneqq \alpha^{\Sigma \leftrightarrow m}(f)(\{x\}) \\ \gamma(f)(\varsigma) &\coloneqq \gamma^{\Sigma \leftrightarrow m}(\lambda\{x_1, \ldots, x_n\}.f(x_1) \sqcup^m \cdots \sqcup^m f(x_n))(\varsigma) \end{aligned}$$

α and γ are monotonic by inspection, and extensive and reductive:

$$\mathit{extensive} : \forall f \varsigma.f(\varsigma) \sqsubseteq \gamma(\alpha(f))(\varsigma)$$

$$\gamma(\alpha(f))(\varsigma)$$

$$= \quad \wr \quad \text{definition of } \alpha \text{ and } \gamma \quad \int$$

$$\gamma^{\Sigma \leftrightarrow m}(\lambda\{x_1, \ldots, x_n\}.\alpha^{\Sigma \leftrightarrow m}(f)(\{x_1\}) \sqcup^m \cdots \sqcup^m \alpha^{\Sigma \leftrightarrow m}(f)(\{x_n\}))(\varsigma)$$

$= \quad \wr \quad$ join-semilattice functorality of $m \quad \int$

$\gamma^{\Sigma\leftrightarrow m}(\lambda\{x_1,\ldots,x_n\}.\alpha^{\Sigma\leftrightarrow m}(f)(\{x_1,\ldots,x_n\}))(\varsigma)$

$\sqsupseteq \quad \wr \quad \gamma^{\Sigma\leftrightarrow m} \circ \alpha^{\Sigma\leftrightarrow m}$ extensive and η-reduction $\quad \int$

$f(\varsigma) \quad \blacksquare$

reductive : $\forall f x.\alpha(\gamma(f))(x) \sqsubseteq f(x)$

$\alpha(\gamma(f))(x)$

$= \quad \wr \quad$ definition of α and $\gamma \quad \int$

$\alpha^{\Sigma\leftrightarrow m}(\gamma^{\Sigma\leftrightarrow m}(\lambda\{x_1,\ldots,x_n\}.f(x_1) \sqcup^m \cdots \sqcup^m f(x_n)))(\{x\})$

$\sqsubseteq \quad \wr \quad \alpha^{\Sigma\leftrightarrow m} \circ \gamma^{\Sigma\leftrightarrow m}$ reductive $\quad \int$

$(\lambda\{x_1,\ldots,x_n\}.f(x_1) \sqcup^m \cdots \sqcup^m f(x_n))(\{x\})$

$= \quad \wr \quad \beta$-reduction $\quad \int$

$f(x) \quad \blacksquare$

Finally, Property (3) commutes:

goal : $\Pi^{\wp^t}(\gamma^{\Sigma\leftrightarrow m}(f))(\varsigma) \sqsubseteq \gamma(\wp^t(f))(\varsigma)$

$\Pi^{\wp^t}(\gamma^{\Sigma\leftrightarrow m}(f))(\varsigma)$

$= \quad \wr \quad$ definition of $\Pi^{\wp^t} \quad \int$

$\gamma^{\Sigma\leftrightarrow m}(\wp^t(\alpha^{\Sigma\leftrightarrow m}(\gamma^{\Sigma\leftrightarrow m}(f))))(\varsigma)$

$\sqsubseteq \quad \wr \quad \alpha^{\Sigma\leftrightarrow m} \circ \gamma^{\Sigma\leftrightarrow m}$ reductive $\quad \int$

$\gamma^{\Sigma\leftrightarrow m}(\wp^t(f))(\varsigma)$

$$= \quad \wr \quad \text{definition of } \gamma \quad \int$$

$$\gamma(\wp^t(f))(\varsigma) \quad \blacksquare$$

FLOW SENSITIVITY *$F^t[s]$ is a Galois transformer.*

Recall the definition of $F^t[s]$ and $\Pi^{F^t}[s]$:

$$\begin{aligned} F^t[s](m)(A) &\coloneqq s \to m([A \mapsto s]) \\ \Pi^{F^t}[s](\Sigma)(A) &\coloneqq \Sigma([A \mapsto s]) \end{aligned}$$

FLOW SENSITIVITY PROPERTY (1) The action $F^t[s]$ on functions:

$$\begin{aligned} F^t[s] &: (A \to m(B)) \to A \to F^t[s](m)(B) \\ F^t[s](f)(x)(s) &\coloneqq y \leftarrow^m f(x)\,;\ \mathit{return}^m(\{y \mapsto s\}) \end{aligned}$$

To transport Galois connections we assume $A \to m_1(B) \underset{\alpha^m}{\overset{\gamma^m}{\leftrightarrows}} A \to m_2(B)$ and define α and γ:

$$\begin{aligned} \alpha &: (A \to F^t[s](m_1)(B)) \to A \to F^t[s](m_2)(B) \\ \gamma &: (A \to F_t[s](m_2)(B)) \to A \to F^t[s](m_1)(B) \end{aligned}$$

$$\begin{aligned} \alpha(f)(x)(s) &\coloneqq \alpha^m(\lambda\{x_1 \mapsto s_1, \ldots, x_n \mapsto s_n\}. \\ &\qquad f(x_1)(s_1) \sqcup^m \cdots \sqcup^m f(x_n)(s_n))(\{x \mapsto s\}) \\ \gamma(f)(x)(s) &\coloneqq \gamma^m(\lambda\{x_1 \mapsto s_1, \ldots, x_n \mapsto s_n\}. \\ &\qquad f(x_1)(s_1) \sqcup^m \cdots \sqcup^m f(x_n)(s_n))(\{x \mapsto s\}) \end{aligned}$$

α and γ are monotonic by inspection. α and γ are extensive and reductive:

$$\mathit{extensive} : \forall f x s. f(x)(s) \sqsubseteq \gamma(\alpha(f))(x)(s)$$

$$\gamma(\alpha(f))(x)(s)$$

$= \quad \wr \quad$ definition of α and $\gamma \quad \int$

$\gamma^m(\lambda\{x_1 \mapsto s_1, \ldots, x_n \mapsto s_n\}.$

$\alpha^m(\lambda\{x_1 \mapsto s_1, \ldots, x_n \mapsto s_n\}.$

$f(x_1)(s_1) \sqcup^{m_1} \cdots \sqcup^{m_1} f(x_n)(s_n))(\{x_1 \mapsto s_1\})$

$\sqcup^{m_2} \cdots \sqcup^{m_2}$

$\alpha^m(\lambda\{x_1 \mapsto s_1, \ldots, x_n \mapsto s_n\}.$

$f(x_1)(s_1) \sqcup^{m_1} \cdots \sqcup^{m_1} f(x_n)(s_n))(\{x_n \mapsto s_n\}))(\{x \mapsto s\})$

$\sqsupseteq \quad \wr \quad$ left-unit of m and $\alpha^m \circ \gamma^m$ reductive $\quad \int$

$\gamma^m(\lambda\{x_1 \mapsto s_1, \ldots, x_n \mapsto s_n\}.$

$(\{x_1 \mapsto s_1, \ldots, x_n \mapsto s_n\} \leftarrow^{m_2} \alpha^m(\gamma^m(\mathit{return}^{m_2}(\{x_1 \mapsto s_1\}))) \,;$

$\alpha^m(f(x_1)(s_1) \sqcup^{m_1} \cdots \sqcup^{m_1} f(x_n)(s_n)))$

$\sqcup^{m_2} \cdots \sqcup^{m_2}$

$(\{x_1 \mapsto s_1, \ldots, x_n \mapsto s_n\} \leftarrow^{m_2} \alpha^m(\gamma^m(\mathit{return}^{m_2}(\{x_n \mapsto s_n\}))) \,;$

$\alpha^m(f(x_1)(s_1) \sqcup^{m_1} \cdots \sqcup^{m_1} f(x_n)(s_n))))(\{x \mapsto s\})$

$= \quad \wr \quad \alpha^m$ and γ^m homomorphic and join functorality $\quad \int$

$\gamma^m(\alpha^m(\lambda\{x_1 \mapsto s_1, \ldots, x_n \mapsto s_n\}.$

$\{x_1 \mapsto s_1, \ldots, x_n \mapsto s_n\} \leftarrow^{m_1} \mathit{return}^{m_1}(\{x_1 \mapsto s_1, \ldots, x_n \mapsto s_n\}) \,;$

$f(x_1)(s_1) \sqcup^{m_1} \cdots \sqcup^{m_1} f(x_n)(s_n)))(\{x \mapsto s\})$

$\sqsupseteq \quad \wr \quad \gamma^m \circ \alpha^m$ extensive and left-unit of $m \quad \int$

$f(x)(s) \quad \blacksquare$

$$\mathit{reductive} \;:\; \forall f x s.\alpha(\gamma(f))(x)(s) \sqsubseteq f(x)(s)$$

$$\alpha(\gamma(f))(x)(s)$$

$$= \quad \wr \quad \text{definition of } \alpha \text{ and } \gamma \quad \int$$

$$\alpha^m(\lambda\{x_1 \mapsto s_1, \ldots, x_n \mapsto s_n\}.$$

$$\gamma^m(\lambda\{x_1 \mapsto s_1, \ldots, x_n \mapsto s_n\}.$$

$$f(x_1)(s_1) \sqcup^{m_2} \cdots \sqcup^{m_2} f(x_n)(s_n))(\{x_1 \mapsto s_1\})$$

$$\sqcup^{m_1} \cdots \sqcup^{m_1}$$

$$\gamma^m(\lambda\{x_1 \mapsto s_1, \ldots, x_n \mapsto s_n\}.$$

$$f(x_1)(s_1) \sqcup^{m_2} \cdots \sqcup^{m_2} f(x_n)(s_n))(\{x_n \mapsto s_n\}))(\{x \mapsto s\})$$

$$\sqsubseteq \quad \wr \quad \text{left-unit of } m \text{ and } \gamma^m \circ \alpha^m \text{ extensive} \quad \int$$

$$\alpha^m(\lambda\{x_1 \mapsto s_1, \ldots, x_n \mapsto s_n\}.$$

$$(\{x_1 \mapsto s_1, \ldots, x_n \mapsto s_n\} \leftarrow^{m_1} \gamma^m(\alpha^m(\mathit{return}^{m_1}(\{x_1 \mapsto s_1\}))) \;;$$

$$\gamma^m(f(x_1)(s_1) \sqcup^{m_2} \cdots \sqcup^{m_2} f(x_n)(s_n)))$$

$$\sqcup^{m_1} \cdots \sqcup^{m_1}$$

$$(\{x_1 \mapsto s_1, \ldots, x_n \mapsto s_n\} \leftarrow^{m_1} \gamma^m(\alpha^m(\mathit{return}^{m_1}(\{x_n \mapsto s_n\}))) \;;$$

$$\gamma^m(f(x_1)(s_1) \sqcup^{m_2} \cdots \sqcup^{m_2} f(x_n)(s_n))))(\{x \mapsto s\})$$

$$= \quad \wr \quad \alpha^m \text{ and } \gamma^m \text{ homomorphic and join functorality} \quad \int$$

$$\alpha^m(\gamma^m(\lambda\{x_1 \mapsto s_1, \ldots, x_n \mapsto s_n\}.$$

$$\{x_1 \mapsto s_1, \ldots, x_n \mapsto s_n\} \leftarrow^{m_2} \mathit{return}^{m_2}(\{x_1 \mapsto s_1, \ldots, x_n \mapsto s_n\}) \;;$$

$$f(x_1)(s_1) \sqcup^{m_2} \cdots \sqcup^{m_2} f(x_n)(s_n)))(\{x \mapsto s\})$$

$\sqsubseteq$ $\wr$ $\alpha^m \circ \gamma^m$ extensive and left-unit of m $\int$

$f(x)(s)$ ■

Finally, Property (1) commutes, assuming that $A \to m_1(B) \underset{\alpha^m}{\overset{\gamma^m}{\leftrightarrows}} A \to m_2(B)$ is homomorphic:

$$
\begin{aligned}
&\mathit{goal} : \forall f s.F^t[s][m_2](\alpha^m(f))(x)(s) = \alpha(F^t[s][m_1](f))(x)(s) \\
&\quad \alpha(F^t[s][m_1](f))(x)(s) \\
&\quad = \ \wr \ \text{definition of } \alpha \text{ and } F^t[s][m_1] \ \int \\
&\quad \alpha^m(\lambda\{x_1 \mapsto s_1, \ldots, x_n \mapsto s_n\}. \\
&\qquad (y \leftarrow^{m_1} f(x)\ ;\ \mathit{return}^{m_1}(y_1)(s_1)) \\
&\qquad \sqcup^{m_1} \ldots \sqcup^{m_1} \\
&\qquad (y \leftarrow^{m_1} f(x)\ ;\ \mathit{return}^{m_1}(y_n)(s_n)))(\{x \mapsto s\}) \\
&\quad = \ \wr \ \text{homomorphic on } \mathit{bind}^{m_1} \text{ and } \mathit{return}^{m_1} \ \int \\
&\quad y \leftarrow^{m_2} \alpha^m(f)(x)\ ;\ \mathit{return}^{m_2}(y)(s) \\
&\quad = \ \wr \ \text{definition of } F^t[s][m_2] \ \int \\
&\quad F^t[s][m_2](\alpha^m(f))(x) \quad \blacksquare
\end{aligned}
$$

Flow Sensitivity Property (2) The action $\Pi^{F^t[s]}$ on functions uses the mapping to monadic functions defined in Property (3):

$$
\begin{aligned}
\Pi^{F^t}[s] \ &: \ (\Sigma(A) \to \Sigma(B)) \to \Pi^{F^t}[s](\Sigma)(A) \to \Pi^{F^t}[s](\Sigma)(B) \\
\Pi^{F^t}[s](f)(\varsigma) \ &\coloneqq \ \gamma^{\Sigma \leftrightarrow \gamma}(F^t[s](\alpha^{\Sigma \leftrightarrow \gamma}(f)))
\end{aligned}
$$

To transport Galois connections, we assume $\Sigma_1(A) \to \Sigma_1(B) \underset{\alpha^\Sigma}{\overset{\gamma^\Sigma}{\leftrightarrows}} \Sigma_2(A) \to \Sigma_2(B)$ and define α and γ as instantiations of α^Σ and γ^Σ:

$$\alpha \ : \ (\Pi^{F^t}[s](\Sigma_1)(A) \to \Pi^{F^t}[s](\Sigma_1)(B)) \to \Pi^{F^t}[s](\Sigma_2)(A) \to \Pi^{F^t}[s](\Sigma_2)(B)$$
$$\gamma \ : \ (\Pi^{F^t}[s](\Sigma_2)(A) \to \Pi^{F^t}[s](\Sigma_2)(B)) \to \Pi^{F^t}[s](\Sigma_1)(A) \to \Pi^{F^t}[s](\Sigma_1)(B)$$

$$\alpha(f)(\varsigma) \ \coloneqq \ \alpha^\Sigma(f)(\varsigma)$$
$$\gamma(f)(\varsigma) \ \coloneqq \ \gamma^\Sigma(f)(\varsigma)$$

Monotonicity, reductive and extensive properties carry over by definition. Finally, Property (2) commutes, assuming that α^Σ and α^m commute with both $\gamma^{\Sigma \leftrightarrow m}$ and $\alpha^{\Sigma \leftrightarrow m}$:

$$\textit{goal} \ : \ \Pi^{F^t}[s][\Sigma_2](\alpha^\Sigma(f))(\varsigma) = \alpha^\Sigma(\Pi^{F^t}[s][\Sigma_1](f))(\varsigma)$$

$$\alpha^\Sigma(\Pi^{F^t}[s][\Sigma_1](f))(\varsigma)$$

$$= \ \wr \ \text{definition of } \Pi^{F^t}[s] \ \int$$

$$\alpha^\Sigma(\gamma^{\Sigma \leftrightarrow \gamma}(F^t[s](\alpha^{\Sigma \leftrightarrow \gamma}(f))))(\varsigma)$$

$$= \ \wr \ \text{definition of } F^t[s] \ \int$$

$$\alpha^\Sigma(\gamma^{\Sigma \leftrightarrow \gamma}(\lambda x.\lambda s.y \leftarrow^{m_1} \alpha^{\Sigma \leftrightarrow \gamma}(f)(x) \ ; \ \textit{return}^{m_1}(\{y \mapsto s\})))(\varsigma)$$

$$= \ \wr \ \alpha^\Sigma \text{ and } \gamma^{\Sigma \leftrightarrow \gamma} \text{ commute} \ \int$$

$$\gamma^{\Sigma \leftrightarrow \gamma}(\alpha^m(\lambda x.\lambda s.y \leftarrow^{m_1} \alpha^{\Sigma \leftrightarrow \gamma}(f)(x) \ ; \ \textit{return}^{m_1}(\{y \mapsto s\})))(\varsigma)$$

$$= \ \wr \ \alpha^m \text{ homomorphic} \ \int$$

$$\gamma^{\Sigma \leftrightarrow \gamma}(\lambda x.\lambda s.y \leftarrow^{m_2} \alpha^m(\alpha^{\Sigma \leftrightarrow \gamma}(f))(x) \ ; \ \textit{return}^{m_2}(\{y \mapsto s\}))(\varsigma)$$

$$= \ \wr \ \alpha^m \text{ and } \alpha^{\Sigma \leftrightarrow \gamma} \text{ commute} \ \int$$

$$\gamma^{\Sigma \leftrightarrow \gamma}(\lambda x.\lambda s.y \leftarrow^{m_2} \alpha^{\Sigma \leftrightarrow \gamma}(\alpha^\Sigma(f))(x) \ ; \ \textit{return}^{m_2}(\{y \mapsto s\}))(\varsigma)$$

$= \quad \wr \quad$ definition of $\Pi^{\wp^t}[\Sigma_2]$ and $\alpha^\Sigma \quad \int$

$\Pi^{\wp^t}[\Sigma_2](\alpha^\Sigma(f))(\varsigma) \quad \blacksquare$

Flow Sensitivity Property (3) Assume a Galois connection:

$$\Sigma(A) \to \Sigma(B) \underset{\alpha^{\Sigma \leftrightarrow m}}{\overset{\gamma^{\Sigma \leftrightarrow m}}{\leftrightarrows}} A \to m(B)$$

The Galois connection between $F^t[s](m)$ and $\Pi^{F^t}[s](\Sigma)$ is:

$$\begin{aligned}
\alpha &: (\Pi^{F^t}[s](\Sigma)(A) \to \Pi^{F^t}[s](\Sigma)(B)) \to A \to F^t[s](m)(B) \\
\gamma &: (A \to F^t[s](m)(B)) \to \Pi^{F^t}[s](\Sigma)(A) \to \Pi^{F^t}[s](\Sigma)(B)
\end{aligned}$$

$$\begin{aligned}
\alpha(f)(x)(s) &\coloneqq \alpha^{\Sigma \leftrightarrow m}(f)(\{x \mapsto s\}) \\
\gamma(f)(\varsigma) &\coloneqq \gamma^{\Sigma \leftrightarrow m}(\lambda\{x_1 \mapsto s_1, \ldots, x_n \mapsto s_n\}.f(x_1)(s_1) \sqcup^m \cdots \sqcup^m f(x_n)(s_n))(\varsigma)
\end{aligned}$$

α and γ are monotonic by inspection. α and γ are extensive and reductive:

$$\textit{extensive} : \forall f \varsigma.f(\varsigma) \sqsubseteq \gamma(\alpha(f))(\varsigma)$$

$\gamma(\alpha(f))(\varsigma)$

$= \quad \wr \quad$ definition of α and $\gamma \quad \int$

$\gamma^{\Sigma \leftrightarrow m}(\lambda\{x_1 \mapsto s_1, \ldots, x_n \mapsto s_n\}.$

$\quad \alpha^{\Sigma \leftrightarrow m}(f)(\{x_1 \mapsto s_1\}) \sqcup^m \cdots \sqcup^m \alpha^{\Sigma \leftrightarrow m}(f)(\{x_n \mapsto s_n\}))(\varsigma)$

$= \quad \wr \quad$ join-semilattice functorality of $m \quad \int$

$\gamma^{\Sigma \leftrightarrow m}(\alpha^{\Sigma \leftrightarrow m}(f))(\varsigma)$

$\sqsupseteq \quad \wr \quad \gamma^{\Sigma \leftrightarrow m} \circ \alpha^{\Sigma \leftrightarrow m}$ extensive $\quad \int$

$f(\varsigma) \quad \blacksquare$

$$\textit{reductive} : \forall f x.\alpha(\gamma(f))(x)(s) \sqsubseteq f(x)(s)$$

$$\alpha(\gamma(f))(x)(s)$$

$$= \quad \wr \quad \text{definition of } \alpha \text{ and } \gamma \quad \int$$

$$\alpha^{\Sigma\leftrightarrow m}(\gamma^{\Sigma\leftrightarrow m}(\lambda\{x_1 \mapsto s_1, \ldots, x_n \mapsto s_n\}.$$

$$f(x_1)(s_1) \sqcup^m \cdots \sqcup^m f(x_n)(s_n)))(\{x \mapsto s\})$$

$$\sqsubseteq \quad \wr \quad \alpha^{\Sigma\leftrightarrow m} \circ \gamma^{\Sigma\leftrightarrow m} \text{ reductive} \quad \int$$

$$(\lambda\{x_1 \mapsto s_1, \ldots, x_n \mapsto s_n\}.f(x_1)(s_1) \sqcup^m \cdots \sqcup^m f(x_n)(s_n))(\{x \mapsto s\})$$

$$= \quad \wr \quad \beta\text{-reduction} \quad \int$$

$$f(x)(s) \quad \blacksquare$$

Finally, Property (3) commutes:

$$\textit{goal} : \Pi^{F^t}[s](\gamma^{\Sigma\leftrightarrow m}(f))(\varsigma) \sqsubseteq \gamma(F^t[s](f))(\varsigma)$$

$$\Pi^{F^t}[s](\gamma^{\Sigma\leftrightarrow m}(f))(\varsigma)$$

$$= \quad \wr \quad \text{definition of } \Pi^{F^t}[s] \quad \int$$

$$\gamma^{\Sigma\leftrightarrow m}(F^t[s](\alpha^{\Sigma\leftrightarrow m}(\gamma^{\Sigma\leftrightarrow m}(f))))(\varsigma)$$

$$\sqsubseteq \quad \wr \quad \alpha^{\Sigma\leftrightarrow m} \circ \gamma^{\Sigma\leftrightarrow m} \text{ reductive} \quad \int$$

$$\gamma^{\Sigma\leftrightarrow m}(F^t[s](f))(\varsigma)$$

$$= \quad \wr \quad \text{definition of } \gamma \quad \int$$

$$\gamma(F^t[s](f))(\varsigma) \quad \blacksquare$$

A.0.2 Lemma 3 [$\mathcal{P}^t$ laws] (Section 5.8.2)

$bind^{\mathcal{P}^t}$ and $return^{\mathcal{P}^t}$ satisfy monad laws, $get^{\mathcal{P}^t}$ and $put^{\mathcal{P}^t}$ satisfy state monad laws, and $mzero^{\mathcal{P}^t}$ and $\boxplus^{\mathcal{P}^t}$ satisfy nondeterminism monad laws:

left-unit : $\forall f x.bind^{\mathcal{P}^t}(return^{\mathcal{P}^t}(x))(f) = f(x)$

$bind^{\mathcal{P}^t}(return^{\mathcal{P}^t}(x))(f)$

$= \wr$ definition of $bind^{\mathcal{P}^t}$ $\int$

$\{x_1, \ldots, x_n\} \leftarrow^m return^{\mathcal{P}^t}(x)\ ;\ f(x_1) \sqcup^m \cdots \sqcup^m f(x_n)$

$= \wr$ definition of $return^{\mathcal{P}^t}$ $\int$

$\{x_1, \ldots, x_n\} \leftarrow^m return^m(\{x\})\ ;\ f(x_1) \sqcup^m \cdots \sqcup^m f(x_n)$

$= \wr$ do-notation for m $\int$

$bind^m(return^m(\{x\}))(\lambda\{x_1, \ldots, x_n\}.f(x_1) \sqcup^m \cdots \sqcup^m f(x_n))$

$= \wr$ left-unit for m $\int$

$f(x)$ ■

right-unit : $\forall X.bind^{\mathcal{P}^t}(X)(return^{\mathcal{P}^t}) = X$

$bind^{\mathcal{P}^t}(X)(return^{\mathcal{P}^t})$

$= \wr$ definition of $bind^{\mathcal{P}^t}$ $\int$

$\{x_1, \ldots, x_n\} \leftarrow^m X\ ;\ return^{\mathcal{P}^t}(x_1) \sqcup^m \cdots \sqcup^m return^{\mathcal{P}^t}(x_n)$

$= \wr$ definition of $return^{\mathcal{P}^t}$ $\int$

$\{x_1, \ldots, x_n\} \leftarrow^m X\ ;\ return^m(\{x_1\}) \sqcup^m \cdots \sqcup^m return^m(\{x_n\})$

$= \quad \wr$ join-semilattice functorality of m distribution over $return^m$ $\int$

$\{x_1, \ldots, x_n\} \leftarrow^m X \; ; \; return^m(\{x_1\} \cup \cdots \cup \{x_n\})$

$= \quad \wr$ definition of $\cup$ $\int$

$\{x_1, \ldots, x_n\} \leftarrow^m X \; ; \; return^m(\{x_1, \ldots, x_n\})$

$= \quad \wr$ do-notation for m $\int$

$bind^m(X)(return^m)$

$= \quad \wr$ right-unit for m $\int$

$X \quad \blacksquare$

$associativity \; : \; \forall f g X. bind^{\mathcal{P}^t}(bind^{\mathcal{P}^t}(X)(f))(g) = bind^{\mathcal{P}^t}(X)(\lambda x. bind^{\mathcal{P}^t}(f(x))(g))$

$bind^{\mathcal{P}^t}(bind^{\mathcal{P}^t}(X)(f))(g)$

$= \quad \wr$ definition of $bind^{\mathcal{P}^t}$ $\int$

$\{y_1, \ldots, y_n\} \leftarrow^m bind^{\mathcal{P}^t}(X)(f) \; ; \; g(y_1) \sqcup^m \cdots \sqcup^m g(y_n)$

$= \quad \wr$ definition of $bind^{\mathcal{P}^t}$ $\int$

$\{y_1, \ldots, y_n\} \leftarrow^m (\{x_1, \ldots, x_n\} \leftarrow^m X \; ; \; f(x_1) \sqcup^m \cdots \sqcup^m f(x_n)) \; ;$

$g(y_1) \sqcup^m \cdots \sqcup^m g(y_n)$

$= \quad \wr$ do-notation for m $\int$

$\{y_1, \ldots, y_n\} \leftarrow^m bind^m(X)(\lambda\{x_1, \ldots, x_n\}. f(x_1) \sqcup^m \cdots \sqcup^m f(x_n)) \; ;$

$g(y_1) \sqcup^m \cdots \sqcup^m g(y_n)$

$=$ $\wr$ do-notation for m $\int$

$bind^m(bind^m(X)(\lambda\{x_1,\ldots,x_n\}.f(x_1) \sqcup^m \cdots \sqcup^m f(x_n)))$

$(\lambda\{y_1,\ldots,y_n\}.g(y_1) \sqcup^m \cdots \sqcup^m g(y_n))$

$=$ $\wr$ associativity for m $\int$

$bind^m(X)(\lambda\{x_1,\ldots,x_n\}.bind^m(f(x_1) \sqcup^m \cdots \sqcup^m f(x_n))$

$(\lambda\{y_1,\ldots,y_n\}.g(y_1) \sqcup^m \cdots \sqcup^m g(y_n)))$

$=$ $\wr$ do-notation for m $\int$

$\{x_1,\ldots,x_n\} \leftarrow^m X$;

$bind^m(f(x_1) \sqcup^m \cdots \sqcup^m f(x_n))(\lambda\{y_1,\ldots,y_n\}.g(y_1) \sqcup^m \cdots \sqcup^m g(y_n))$

$=$ $\wr$ do-notation for m $\int$

$\{x_1,\ldots,x_n\} \leftarrow^m X$; $\{y_1,\ldots,y_n\} \leftarrow^m (f(x_1) \sqcup^m \cdots \sqcup^m f(x_n))$;

$g(y_1) \sqcup^m \cdots \sqcup^m g(y_n)$

$=$ $\wr$ join-semilattice functorality of m distribution over $bind^m$ $\int$

$\{x_1,\ldots,x_n\} \leftarrow^m X$;

$(\{y_1,\ldots,y_n\} \leftarrow^m f(x_1)$; $g(y_1) \sqcup^m \cdots \sqcup^m g(y_n))$

$\sqcup^m \cdots \sqcup^m$

$(\{y_1,\ldots,y_n\} \leftarrow^m f(x_n)$; $g(y_1) \sqcup^m \cdots \sqcup^m g(y_n))$

$=$ $\wr$ definition of $bind^{\mathcal{P}^t}$ $\int$

$\{x_1,\ldots,x_n\} \leftarrow^m X$; $bind^{\mathcal{P}^t}(f(x_1))(g) \sqcup^m \cdots \sqcup^m bind^{\mathcal{P}^t}(f(x_n)(g))$

$= \quad \wr$ definition of $bind^{\mathcal{P}^t} \quad \int$

$bind^{\mathcal{P}^t}(X)(\lambda x.bind^{\mathcal{P}^t}(f(x))(g))$ ■

get-get : $s_1 \leftarrow get^{\mathcal{P}^t}\ ;\ s_2 \leftarrow get^{\mathcal{P}^t}\ ;\ return^{\mathcal{P}^t}(s_1, s_2) = s \leftarrow get^{\mathcal{P}^t}\ ;\ return^{\mathcal{P}^t}(s, s)$

$s_1 \leftarrow get^{\mathcal{P}^t}\ ;\ s_2 \leftarrow get^{\mathcal{P}^t}\ ;\ return^{\mathcal{P}^t}(s_1, s_2)$

$= \quad \wr$ definition of $get^{\mathcal{P}^t} \quad \int$

$s_1 \leftarrow (s \leftarrow^m get^m\ ;\ return(\{s\}))\ ;$

$s_2 \leftarrow (s \leftarrow get^m\ ;\ return(\{s\}))\ ;\ return^{\mathcal{P}^t}(s_1, s_2)$

$= \quad \wr$ definition of $return^{\mathcal{P}^t} \quad \int$

$s_1 \leftarrow (s \leftarrow^m get^m\ ;\ return(\{s\}))\ ;$

$s_2 \leftarrow (s \leftarrow get^m\ ;\ return(\{s\}))\ ;\ return^m(\{\langle s_1, s_2\rangle\})$

$= \quad \wr$ do-notation for m and definition of $bind^{\mathcal{P}^t} \quad \int$

$\{s_{11}, \ldots, s_{1n}\} \leftarrow^m (s \leftarrow^m get^m\ ;\ return(\{s\}))\ ;$

$(s_2 \leftarrow (s \leftarrow^m get^m\ ;\ return(\{s\}))\ ;\ return^m(\{\langle s_{11}, s_2\rangle\}))$

$\sqcup^m \cdots \sqcup^m$

$(s_2 \leftarrow (s \leftarrow^m get^m\ ;\ return(\{s\}))\ ;\ return^m(\{\langle s_{1n}, s_2\rangle\}))$

$= \quad \wr$ associativity and left-unit of $m \quad \int$

$s_1 \leftarrow^m get^m\ ;\ (s_2 \leftarrow (s \leftarrow^m get^m\ ;\ return(\{s\}))\ ;\ return^m(\{\langle s_1, s_2\rangle\}))$

$= \quad \wr$ do-notation for m and definition of $bind^{\mathcal{P}^t} \quad \int$

$s_1 \leftarrow^m get^m \; ;$

$\{s_{21}, \ldots, s_{2n}\} \leftarrow^m (s \leftarrow^m get^m \; ; \; return(\{s\})) \; ;$

$return^m(\{\langle s_1, s_{21}\rangle\}) \sqcup^m \cdots \sqcup^m return^m(\{\langle s_1, s_{2n}\rangle\})$

$= \quad \wr$ associativity and left-unit of $m \quad \int$

$s_1 \leftarrow^m get^m \; ; \; s_2 \leftarrow^m get^m \; ; \; return^m(\{\langle s_1, s_2\rangle\})$

$= \quad \wr$ associativity and left-unit of $m \quad \int$

$p \leftarrow^m (s_1 \leftarrow^m get^m \; ; \; s_2 \leftarrow^m get^m \; ; \; return^m(s_1, s_2)) \; ; \; return^m(\{p\})$

$= \quad \wr$ get-get of $m \quad \int$

$p \leftarrow^m (s \leftarrow get^m \; ; \; return^m(s, s)) \; ; \; return^m(\{p\})$

$= \quad \wr$ associativity and left-unit of $m \quad \int$

$s \leftarrow^m get^m \; ; \; return^m(\{\langle s, s\rangle\})$

$= \quad \wr$ associativity and left-unit of $m \quad \int$

$\{s_1, \ldots, s_n\} \leftarrow^m (s \leftarrow^m get^m \; ; \; return^m(\{s\})) \; ;$

$return^m(\{\langle s_1, s_1\rangle\}) \sqcup^m \cdots \sqcup^m return^m(\{\langle s_n, s_n\rangle\})$

$= \quad \wr$ definition of $get^{\mathcal{P}^t}$ and $return^{\mathcal{P}^t} \quad \int$

$s \leftarrow get^{\mathcal{P}^t} \; ; \; return^{\mathcal{P}^t}(s, s) \quad \blacksquare$

get-put : $(s \leftarrow get^{\mathcal{P}^t} \; ; \; put^{\mathcal{P}^t}(s)) = return(\bullet)$

put-get : $\forall s.(\bullet \leftarrow put^{\mathcal{P}^t}(s) \; ; \; get^{\mathcal{P}^t}) = (\bullet \leftarrow put^{\mathcal{P}^t}(s) \; ; \; return^{\mathcal{P}^t}(s))$

put-put : $\forall s_1 s_2.(\bullet \leftarrow put^{\mathcal{P}^t}(s_1) \; ; \; put^{\mathcal{P}^t}(s_2)) = put^{\mathcal{P}^t}(s_2)$

get-put, *put-get* and *put-put* are analogous to *get-get*; they follow from monad associativity and the property from the underlying monad.

$$\textit{mzero-unit} \ : \ \forall X. \textit{mzero}^{\mathcal{P}^t} \boxplus^{\mathcal{P}^t} X = X$$

$$\boxplus\textit{-associativity} \ : \ \forall XYZ.(X \boxplus^{\mathcal{P}^t} Y) \boxplus^{\mathcal{P}^t} Z \boxplus^{\mathcal{P}^t} = X \boxplus^{\mathcal{P}^t} (Y \boxplus^{\mathcal{P}^t} Z)$$

$$\boxplus\textit{-commutativity} \ : \ \forall XY. X \boxplus^{\mathcal{P}^t} Y = Y \boxplus^{\mathcal{P}^t} X$$

$$\boxplus\textit{-idempotence} \ : \ \forall X. X \boxplus^{\mathcal{P}^t} X = X$$

$$\textit{mzero-left-zero} \ : \ \forall k.(x \leftarrow \textit{mzero}^{\mathcal{P}^t} \ ; \ k(x)) = \textit{mzero}^{\mathcal{P}^t}$$

$$\textit{mzero-right-zero} \ : \ \forall X.(x \leftarrow X \ ; \ \textit{mzero}^{\mathcal{P}^t}) = \textit{mzero}^{\mathcal{P}^t}$$

$$\boxplus\textit{-distributivity} \ : \ \forall XYk.$$

$$(x \leftarrow X \boxplus^{\mathcal{P}^t} Y \ ; \ k(x)) = (x \leftarrow X \ ; \ k(x)) \boxplus^{\mathcal{P}^t} (x \leftarrow Y \ ; \ k(x))$$

These follow directly from the definition of $\textit{mzero}^{\mathcal{P}^t}$ and $\boxplus^{\mathcal{P}^t}$ and the join-semilattice properties from the underlying monad.

A.0.3 Lemma 4 [F^t laws] (Section 5.8.3)

$\textit{bind}^{F^t}$ and $\textit{return}^{F^t}$ satisfy monad laws, $\textit{get}^{F^t}$ and $\textit{put}^{F^t}$ satisfy state monad laws, and $\textit{mzero}^{F^t}$ and $\boxplus^{F^t}$ satisfy nondeterminism monad laws.

We go into slightly less detail in these proofs than was done for $\mathcal{P}^t$.

$$\textit{left-unit} \ : \ \forall fxs. \textit{bind}^{F^t}(\textit{return}^{F^t}(x))(f)(s) = f(x)(s)$$

$$\textit{bind}^{F^t}(\textit{return}^{F^t}(x))(f)(s)$$

$= \quad \wr$ definition of $bind^{F^t}$ and $return^{F^t} \quad \int$

$\{x_1 \mapsto s_1, \ldots, x_n \mapsto s_n\} \leftarrow^m return^m(\{x \mapsto s\})\ ;\ f(x_1)(s_1) \sqcup^m \cdots \sqcup^m f(x_n)(s_n)$

$= \quad \wr$ left-unit for $m \quad \int$

$f(x)(s) \quad \blacksquare$

right-unit : $\forall X s.bind^{F^t}(X)(return^{F^t})(s) = X(s)$

$bind^{F^t}(X)(return^{F^t})(s)$

$= \quad \wr$ definition of $bind^{F^t}$ and $return^{F^t} \quad \int$

$\{x_1 \mapsto s_1, \ldots, x_n \mapsto s_n\} \leftarrow^m X(s)\ ;$

$return^m(\{x_1 \mapsto s_1\}) \sqcup^m \cdots \sqcup^m return^m(\{x_n \mapsto s_n\})$

$= \quad \wr$ join-semilattice functorality of m distribution over $return^m \quad \int$

$\{x_1 \mapsto s_1, \ldots, x_n \mapsto s_n\} \leftarrow^m X(s)\ ;\ return^m(\{x_1 \mapsto s_1, \ldots, x_n \mapsto s_n\})$

$= \quad \wr$ right-unit of $m \quad \int$

$X(s) \quad \blacksquare$

associativity : $\forall f g X s.$

$bind^{F^t}(bind^{F^t}(X)(f))(g)(s) = bind^{F^t}(X)(\lambda x.bind^{F^t}(f(x))(g))(s)$

$bind^{F^t}(bind^{F^t}(X)(f))(g)(s)$

$= \quad \wr$ definition of $bind^{F^t} \quad \int$

$\{y_1 \mapsto s_{y1}, \ldots, y_n \mapsto s_{yn}\} \leftarrow^m$

$(\{x_1 \mapsto s_{x1}, \ldots, x_n \mapsto s_{xn}\} \leftarrow X(s)\ ;\ f(x_1)(s_{x1}) \sqcup^m \cdots \sqcup^m f(x_n)(s_{xn}))\ ;$

$g(y_1)(s_{y1}) \sqcup^m \cdots \sqcup^m g(y_n)(s_{yn})$

$= \quad \wr$ associativity of m $\int$

$\{x_1 \mapsto s_{x1}, \ldots, x_n \mapsto s_{xn}\} \leftarrow^m X(s)$;

$\{y_1 \mapsto s_{y1}, \ldots, y_n \mapsto s_{yn}\} \leftarrow^m f(x_1)(s_{x1}) \sqcup^m \cdots \sqcup^m f(x_n)(s_{xn})$;

$g(y_1)(s_{y1}) \sqcup^m \cdots \sqcup^m g(y_n)(s_{yn})$

$= \quad \wr$ join-semilattice functorality of m distribution over $bind^m$ $\int$

$\{x_1 \mapsto s_{x1}, \ldots, x_n \mapsto s_{xn}\} \leftarrow^m X(s)$;

$(\{y_1 \mapsto s_{y1}, \ldots, y_n \mapsto s_{yn}\} \leftarrow^m f(x_1)(s_{x1})$; $g(y_1)(s_{y1}) \sqcup^m \cdots \sqcup^m g(y_n)(s_{yn}))$

$\sqcup^m \cdots \sqcup^m$

$(\{y_1 \mapsto s_{y1}, \ldots, y_n \mapsto s_{yn}\} \leftarrow^m f(x_n)(s_{xn})$; $g(y_1)(s_{y1}) \sqcup^m \cdots \sqcup^m g(y_n)(s_{yn}))$

$= \quad \wr$ definition of $bind^{F^t}$ $\int$

$bind^{F^t}(X)(\lambda x.bind^{F^t}(f(x))(g))(s)$ ■

get-get : $\forall s.$

$(s_1 \leftarrow get^{F^t}$; $s_2 \leftarrow get^{F^t}$; $return^{F^t}(s_1, s_2))(s) = (s \leftarrow get^{F^t}$; $return^{F^t}(s, s))(s)$

$(s_1 \leftarrow get^{F^t}$; $s_2 \leftarrow get^{F^t}$; $return^{F^t}(s_1, s_2))(s)$

$= \quad \wr$ definition of $bind^{F^t}$ and get^{F^t} $\int$

$\{x_1 \mapsto s_{x1}, \ldots, x_n \mapsto s_{xn}\} \leftarrow^m return^m\{s \mapsto s\}$;

$(s_2 \leftarrow get^{F^t}$; $return^{F^t}(x_1, s_2))(s_{x1})$

$\sqcup^m \cdots \sqcup^m$

$(s_2 \leftarrow get^{F^t}$; $return^{F^t}(x_n, s_2))(s_{xn})$

$= \quad \wr \quad$ left-unit of m $\quad \int$

$(s_2 \leftarrow \mathit{get}^{F^t} \; ; \; \mathit{return}^{F^t}(s, s_2))(s)$

$= \quad \wr \quad$ definition of bind^{F^t} and $\mathit{get}^{F^t} \quad \int$

$\{x_1 \mapsto s_{x1}, \ldots, x_n \mapsto s_{xn}\} \leftarrow^m \mathit{return}^m \{s \mapsto s\} \; ;$

$\mathit{return}^{F^t}(s, x_1)(s_{x1}) \sqcup^m \cdots \sqcup^m \mathit{return}^{F^t}(s, x_n)(s_{xn})$

$= \quad \wr \quad$ left-unit of m $\quad \int$

$\mathit{return}^{F^t}(s, s)(s)$

$= \quad \wr \quad$ left-unit of m and definition of $\mathit{get}^{F^t} \quad \int$

$(s \leftarrow \mathit{get}^{F^t} \; ; \; \mathit{return}^{F^t}(s, s))(s) \quad \blacksquare$

get-put : $\forall s.(s_1 \leftarrow \mathit{get}^{F^t} \; ; \; \mathit{put}^{F^t}(s_1))(s) = \mathit{return}(\bullet)(s)$

$(s_1 \leftarrow \mathit{get}^{F^t} \; ; \; \mathit{put}^{F^t}(s_1))(s)$

$= \quad \wr \quad$ definition of bind^{F^t} and $\mathit{get}^{F^t} \quad \int$

$\{x_1 \mapsto s_{x1}, \ldots, x_n \mapsto s_{xn}\} \leftarrow^m \mathit{return}^m(\{s \mapsto s\})$

$; \; \mathit{put}^{F^t}(x_1)(s_{x1}) \sqcup^m \cdots \sqcup^m \mathit{put}^{F^t}(x_n)(s_{xn})$

$= \quad \wr \quad$ right-unit of m $\quad \int$

$\mathit{put}^{F^t}(s)(s)$

$= \quad \wr \quad$ definition of $\mathit{put}^{F^t} \quad \int$

$\mathit{return}^m(\{\bullet \mapsto s\})$

$= \quad \wr \quad$ definition of $\mathit{return}^{F^t} \quad \int$

$\mathit{return}^{F^t}(\bullet)(s) \quad \blacksquare$

$put\text{-}get \ : \ \forall s s_1.(\bullet \leftarrow put^{F^t}(s_1) \ ; \ get^{F^t})(s) = (\bullet \leftarrow put^{F^t}(s_1) \ ; \ return^{F^t}(s_1))(s)$

$(\bullet \leftarrow put^{F^t}(s_1) \ ; \ get^{F^t})(s)$

$= \ \wr$ definition of $bind^{F^t}$ and $put^{F^t} \ \int$

$\{\bullet \mapsto s\} \leftarrow^m return^m\{\bullet \mapsto s_1\} \ ; \ get^{F^t}(s)$

$= \ \wr$ right-unit of $m \ \int$

$get^{F^t}(s_1)$

$= \ \wr$ definition of get^{F^t} and $return^{F^t} \ \int$

$return^{F^t}(s_1)(s_1)$

$= \ \wr$ definition of $bind^{F^t}$ and $put^{F^t} \ \int$

$(\bullet \leftarrow put^{F^t}(s_1) \ ; \ return^{F_t}(s_1))(s) \quad \blacksquare$

$put\text{-}put \ : \ \forall s s_1 s_2.(\bullet \leftarrow put^{F^t}(s_1) \ ; \ put^{F^t}(s_2))(s) = put^{F^t}(s_2)(s)$

$(\bullet \leftarrow put^{F_t}(s_1) \ ; \ put^{F^t}(s_2))(s)$

$= \ \wr$ definition of $bind^{F^t}$ and $put^{F^t} \ \int$

$\{\bullet \mapsto s\} \leftarrow^m return^m\{\bullet \mapsto s_1\} \ ; \ return^m\{\bullet \mapsto s_2\}$

$= \ \wr$ right-unit of $m \ \int$

$return^m\{\bullet \mapsto s_2\}$

$= \ \wr$ definition of $put^{F^t} \ \int$

$put^{F^t}(s_2)(s) \quad \blacksquare$

$mzero\text{-}unit \ : \ \forall X.mzero^{F^t} \boxplus^{F^t} X = X$

$\boxplus\text{-}associativity \ : \ \forall X Y Z.(X \boxplus^{F^t} Y) \boxplus^{F^t} Z \boxplus^{F^t} = X \boxplus^{F^t} (Y \boxplus^{F^t} Z)$

$\boxplus$-*commutativity* : $\forall XY.X \boxplus^{F^t} Y = Y \boxplus^{F^t} X$

$\boxplus$-*idempotence* : $\forall X.X \boxplus^{F^t} X = X$

mzero-left-zero : $\forall k.(x \leftarrow mzero^{F^t} \ ; \ k(x)) = mzero^{F^t}$

mzero-right-zero : $\forall X.(x \leftarrow X \ ; \ mzero^{F^t}) = mzero^{F^t}$

$\boxplus$-*distributivity* : $\forall XYk.$

$$(x \leftarrow X \boxplus^{F^t} Y \ ; \ k(x)) = (x \leftarrow X \ ; \ k(x)) \boxplus^{F^t} (x \leftarrow Y \ ; \ k(x))$$

These follow directly from the definition of $mzero^{F^t}$ and $\boxplus^{F^t}$ and the join-semilattice properties from the underlying monad.

Bibliography

Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. A functional correspondence between monadic evaluators and abstract machines for languages with computational effects. In *Theoretical Computer Science (TCS)*. Elsevier Science Publishers Ltd., Essex, UK, 2005.

Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

J. W. Backus. The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM conference. In *International Conference on Information Processsing (ICIP)*. UNESCO, Paris, France, 1959.

Gilles Barthe, David Pichardie, and Tamara Rezk. A certified lightweight non-interference Java bytecode verifier. In *European Symposium on Programming (ESOP)*. Springer-Verlag, Berlin, Heidelberg, 2007.

Richard Bird and Oege de Moor. *The Algebra of Programming*. Prentice Hall, Upper Saddle River, NJ, USA, 1996.

Richard S. Bird. A calculus of functions for program derivation. In *Research Topics in Functional Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome 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)*. ACM, New York, NY, USA, 2003.

Sandrine Blazy, Vincent Laporte, André Maroneze, and David Pichardie. Formal verification of a C value analysis based on abstract interpretation. In *Static Analysis Symposium (SAS)*. Springer-Verlag, Berlin, Heidelberg, 2013.

Roland Bol and Lars Degerstedt. Tabulated resolution for well founded semantics. In *International Logic Programming Symposium (ILPS)*. MIT Press, Cambridge, MA, USA, 1993.

David Cachera and David Pichardie. A certified denotational abstract interpreter. In *Interactive Theorem Proving (ITP)*. Springer-Verlag, Berlin, Heidelberg, 2010.

David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *Programming Language Design and Implementation (PLDI)*. ACM, New York, NY, USA, 1990.

Weidong Chen and David S. Warren. Tabled evaluation with delaying for general logic programs. In *Journal of the ACM (JACM)*. ACM, New York, NY, USA, 1996.

Thierry Coquand and Gerard Huet. The calculus of constructions. In *Information and Computation: Semantics of Data Types*. Academic Press, Inc., Duluth, MN, USA, 1988.

Thierry Coquand and Gérard P. Huet. Constructions: A higher order proof system for mechanizing mathematics. In *European Conference on Computer Algebra (EUROCAL)*. Springer-Verlag, London, UK, 1985.

Thierry Coquand and Christine Paulin. Inductively defined types. In *International Conference on Computer Logic (COLOG)*. Springer-Verlag, London, UK, 1990.

Patrick Cousot. The calculational design of a generic abstract interpreter. In *Calculational System Design*, NATO ASI Series F. IOS Press, Amsterdam, The Netherlands, 1999.

Patrick Cousot. Abstract interpretation. MIT Course 16.399, 2005. URL http://web.mit.edu/16.399/www/.

Patrick Cousot. Abstract interpretation, 2008. URL http://www.di.ens.fr/~cousot/AI/.

Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of programs. In *International Symposium on Programming (ISOP)*. Dunod, Paris, France, 1976.

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)*. ACM, New York, NY, USA, 1977.

Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Principles of Programming Languages (POPL)*. ACM, New York, NY, USA, 1979.

Patrick Cousot and Radhia Cousot. Inductive definitions, semantics and abstract interpretations. In *Principles of Programming Languages (POPL)*. ACM, New York, NY, USA, 1992.

Patrick Cousot and Radhia Cousot. Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and PER analysis of functional languages), invited paper. In *International Conference on Computer Languages (ICCL)*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1994.

Patrick Cousot and Radhia Cousot. A Galois connection calculus for abstract interpretation. In *Principles of Programming Languages (POPL)*. ACM, New York, NY, USA, 2014.

David Darais and David Van Horn. Constructive Galois connections: Taming the Galois connection framework for mechanized metatheory. In *International Conference on Functional Programming (ICFP)*. ACM, New York, NY, USA, 2016.

David Darais, Matthew Might, and David Van Horn. Galois transformers and modular abstract interpreters: Reusable metatheory for program analysis. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. ACM, New York, NY, USA, 2015.

David Darais, Nicholas Labich, Phúc C. Nguyễn, and David Van Horn. Definitional abstract interpreters for higher-order programming languages. In *International Conference on Functional Programming (ICFP)*. ACM, New York, NY, USA, 2017.

Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: Path-sensitive program verification in polynomial time. In *Programming Language Design and Implementation (PLDI)*. ACM, New York, NY, USA, 2002.

Steven Dawson, C. R. Ramakrishnan, and David S. Warren. Practical program analysis using general purpose logic programming systems—a case study. In *Programming Language Design and Implementation (PLDI)*. ACM, New York, NY, USA, 1996.

Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. Fiat: Deductive synthesis of abstract data types in a proof assistant. In *Principles of Programming Languages (POPL)*. ACM, New York, NY, USA, 2015.

The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004.

Christopher Earl. *Introspective Pushdown Analysis and Nebo*. PhD thesis, University of Utah, 2014.

Christopher Earl, Matthew Might, and David Van Horn. Pushdown control-flow analysis of higher-order programs. In *Workshop on Scheme and Functional Programming (Scheme)*, 2010.

Christopher Earl, Ilya Sergey, Matthew Might, and David Van Horn. Introspective pushdown analysis of higher-order programs. In *International Conference on Functional Programming (ICFP)*. ACM, New York, NY, USA, 2012.

Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. In *Theoretical Computer Science (TCS)*. Elsevier Science Publishers Ltd., Essex, UK, 1992.

Mattias Felleisen and Daniel P. Friedman. A calculus for assignments in higher-order languages. In *Principles of Programming Languages (POPL)*. ACM, New York, NY, USA, 1987.

Mattias Felleisen, Daniel P. Friedman, Eugene Kohlbecker, and Bruce Duba. A syntactic theory of sequential control. In *Theoretical Computer Science (TCS)*. Elsevier Science Publishers Ltd., Essex, UK, 1987.

Sebastian Fischer, Oleg Kiselyov, and Chung-chieh Shan. Purely functional lazy non-deterministic programming. In *International Conference on Functional Programming (ICFP)*. ACM, New York, NY, USA, 2009.

Matthew Flatt and Matthias Felleisen. Units: Cool modules for HOT languages. In *Programming Language Design and Implementation (PLDI)*. ACM, New York, NY, USA, 1998.

Matthew Flatt and PLT. Reference: Racket. Technical report, PLT Design Inc., 2010. URL https://racket-lang.org/tr1/.

Ronald Garcia, Alison M. Clark, and Éric Tanter. Abstracting gradual typing. In *Principles of Programming Languages (POPL)*. ACM, New York, NY, USA, 2016.

Jeremy Gibbons and Ralf Hinze. Just do it: Simple monadic equational reasoning. In *International Conference on Functional Programming (ICFP)*. ACM, New York, NY, USA, 2011.

Thomas Gilray, Michael D. Adams, and Matthew Might. Allocation characterizes polyvariance: A unified methodology for polyvariant control-flow analysis. In *International Conference on Functional Programming (ICFP)*. ACM, New York, NY, USA, 2016a.

Thomas Gilray, Steven Lyde, Michael D. Adams, Matthew Might, and David Van Horn. Pushdown control-flow analysis for free. In *Principles of Programming Languages (POPL)*. ACM, New York, NY, USA, 2016b.

Robert Glück. Simulation of two-way pushdown automata revisited. In *Electronic Proceedings in Theoretical Computer Science (EPTCS)*, volume Semantics, Abstract Interpretation, and Reasoning about Programs: Essays Dedicated to David A. Schmidt on the Occasion of his Sixtieth Birthday (Festschrift for Dave Schmidt). Open Publishing Association, 2013.

Ben Hardekopf, Ben Wiedermann, Berkeley Churchill, and Vineeth Kashyap. Widening for control-flow. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*. Springer-Verlag New York, Inc., New York, NY, USA, 2014.

Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *Program Analysis for Software Tools and Engineering (PASTE)*. ACM, New York, NY, USA, 2001.

Ralf Hinze. Deriving backtracking monad transformers. In *International Conference on Functional Programming (ICFP)*. ACM, New York, NY, USA, 2000.

Suresh Jagannathan and Stephen Weeks. A unified treatment of flow analysis in higher-order languages. In *Principles of Programming Languages (POPL)*. ACM, New York, NY, USA, 1995.

Suresh Jagannathan, Peter Thiemann, Stephen Weeks, and Andrew Wright. Single and loving it: Must-alias analysis for higher-order languages. In *Principles of Programming Languages (POPL)*. ACM, New York, NY, USA, 1998.

Gerda Janssens and Konstantinos Sagonas. On the use of tabling for abstract interpretation: An experiment with abstract equation systems. In *Tabulation in Parsing and Deduction (TAPD)*, 1998.

Mauro Javier Jaskelioff. *Lifting of Operations in Modular Monadic Semantics*. PhD thesis, University of Nottingham, 2009.

James Ian Johnson and David Van Horn. Abstracting abstract control. In *Symposium on Dynamic Languages (DLS)*. ACM, New York, NY, USA, 2014.

James Ian Johnson, Ilya Sergey, Christopher Earl, Matthew Might, and David Van Horn. Pushdown flow analysis with abstract garbage collection. In *Journal of Functional Programming (JFP)*. Cambridge University Press, Cambridge, UK, 2014.

Neil D. Jones. Flow analysis of lambda expressions (preliminary version). In *International Colloquium on Automata, Languages and Programming (ICALP)*. Springer-Verlag, London, UK, 1981.

Neil D. Jones and Flemming Nielson. Abstract interpretation: A semantics-based tool for program analysis. In *Handbook of Logic in Computer Science*. Oxford University Press, Oxford, UK, 1995.

Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. A formally-verified C static analyzer. In *Principles of Programming Languages (POPL)*. ACM, New York, NY, USA, 2015.

George Kastrinis and Yannis Smaragdakis. Hybrid context-sensitivity for points-to analysis. In *Programming Language Design and Implementation (PLDI)*. ACM, New York, NY, USA, 2013.

James C. King. Symbolic execution and program testing. In *Communications of the ACM (CACM)*. ACM, New York, NY, USA, 1976.

Oleg Kiselyov. Typed tagless final interpreters. In *Spring School Conference on Generic and Indexed Programming (SSGIP)*. Springer-Verlag, Berlin, Heidelberg, 2010.

Oleg Kiselyov, Chung-chieh Shan, Daniel P. Friedman, and Amr Sabry. Backtracking, interleaving, and terminating monad transformers: (functional pearl). In *International Conference on Functional Programming (ICFP)*. ACM, New York, NY, USA, 2005.

Xavier Leroy. Formal verification of a realistic compiler. In *Communications of the ACM (CACM)*. ACM, New York, NY, USA, 2009.

Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Principles of Programming Languages (POPL)*. ACM, New York, NY, USA, 1995.

Gregory Malecha and Jesper Bengtson. Extensible and efficient automation through reflective tactics. In *Programming Languages and Systems (PLAS)*. Springer-Verlag New York, Inc., New York, NY, USA, 2016.

Per Martin-Löf. An intuitionistic theory of types: Predicative part. In *Studies in Logic and the Foundations of Mathematics (SLFM)*. Elsevier, Amsterdam, The Netherlands, 1975.

Per Martin-Löf. Intuitionistic type theory. In *Studies in Proof Theory*. Bibliopolis, Naples, Italy, 1984.

Jan Midtgaard. Control-flow analysis of functional programs. In *ACM Computing Surveys (CSUR)*. ACM, New York, NY, USA, 2012.

Jan Midtgaard and Thomas Jensen. A calculational approach to control-flow analysis by abstract interpretation. In *Static Analysis Symposium (SAS)*. Springer-Verlag, Berlin, Heidelberg, 2008.

Jan Midtgaard and Thomas P. Jensen. Control-flow analysis of function calls and returns by abstract interpretation. In *International Conference on Functional Programming (ICFP)*. ACM, New York, NY, USA, 2009.

Matthew Might. *Environment Analysis of Higher-order Languages*. PhD thesis, Georgia Institute of Technology, 2007a.

Matthew Might. Logic-flow analysis of higher-order programs. In *Principles of Programming Languages (POPL)*. ACM, New York, NY, USA, 2007b.

Matthew Might and Olin Shivers. Improving flow analyses via γCFA: Abstract garbage collection and counting. In *International Conference on Functional Programming (ICFP)*. ACM, New York, NY, USA, 2006a.

Matthew Might and Olin Shivers. Environment analysis via δCFA. In *Principles of Programming Languages (POPL)*. ACM, New York, NY, USA, 2006b.

Matthew Might and David Van Horn. Family of abstract interpretations for static analysis of concurrent higher-order programs. In *Static Analysis Symposium (SAS)*. Springer-Verlag, Berlin, Heidelberg, 2011.

Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to analysis for Java. In *Transactions on Software Engineering and Methodology (TOSEM)*. ACM, New York, NY, USA, 2005.

Antoine Miné. The octagon abstract domain. In *Higher Order and Symbolic Computation (HOSC)*. Kluwer Academic Publishers, Dordrecht, The Netherlands, 2006.

Eugenio Moggi. An abstract view of programming languages. Technical report, University of Edinburgh, 1989.

David Monniaux. Réalisation mécanisée d'interpréteurs abstraits. Rapport de DEA, Université Paris VII, 1998. In French.

Phúc C. Nguyễn and David Van Horn. Relatively complete counterexamples for higher-order programs. In *Programming Language Design and Implementation (PLDI)*. ACM, New York, NY, USA, 2015.

Flemming Nielson and Hanne Riis Nielson. Infinitary control flow analysis: A collecting semantics for closure analysis. In *Principles of Programming Languages (POPL)*. ACM, New York, NY, USA, 1997.

Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, Berlin, Heidelberg, 1999.

Ulf Norell. *Towards a Practical Programming Language Based on Dependent Type Theory*. PhD thesis, Chalmers University of Technology, 2007.

David Pichardie. *Interprétation Abstraite en Logique Intuitionniste: Extraction d'Analyseurs Java Certifiés*. PhD thesis, Université Rennes 1, 2005. In French.

Atze van der Ploeg and Oleg Kiselyov. Reflection without remorse: Revealing a hidden sequence to speed up monadic reflection. In *Haskell Symposium (Haskell)*. ACM, New York, NY, USA, 2014.

Gordon D. Plotkin. A structural approach to operational semantics. Technical report, Aarhus University, 1981.

Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Principles of Programming Languages (POPL)*. ACM, New York, NY, USA, 1995.

John C. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM Annual Conference (ACM)*. ACM, New York, NY, USA, 1972.

Ilya Sergey, Jan Midtgaard, and Dave Clarke. Calculating graph algorithms for dominance and shortest path. In *Mathematics of Program Construction (MPC)*. Springer-Verlag, Berlin, Heidelberg, 2012.

Ilya Sergey, Dominique Devriese, Matthew Might, Jan Midtgaard, David Darais, Dave Clarke, and Frank Piessens. Monadic abstract interpreters. In *Programming Language Design and Implementation (PLDI)*. ACM, New York, NY, USA, 2013.

Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*. Prentice Hall, Upper Saddle River, NJ, USA, 1981.

Olin Grigsby Shivers. *Control-Flow Analysis of Higher-Order Languages or Taming Lambda*. PhD thesis, Carnige-Mellon Univeristy, 1991.

Paulo F. Silva and José N. Oliveira. Galculator: Functional prototype of a Galois-connection based proof assistant. In *Principles and Practice of Declarative Programming (PPDP)*. ACM, New York, NY, USA, 2008.

Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well: Understanding object-sensitivity. In *Principles of Programming Languages (POPL)*. ACM, New York, NY, USA, 2011.

Guy L. Steele, Jr. Building interpreters by composing monads. In *Principles of Programming Languages (POPL)*. ACM, New York, NY, USA, 1994.

Terrance Swift and David S. Warren. XSB: Extending prolog with tabled logic programming. In *Theory and Practice of Logic Programming (TPLP)*. Cambridge University Press, Cambridge, UK, 2012.

Hisao Tamaki and Taisuke Sato. OLD resolution with tabulation. In *International Conference on Logic Programming (ICLP)*. Springer-Verlag, London, UK, 1986.

Gregory Tassey. *The Economic Impacts of Inadequate Infrastructure for Software Testing*. National Institute Of Standards and Technology, Gaithersburg, MD, USA, 2002.

Julien Tesson, Hideki Hashimoto, Zhenjiang Hu, Frédéric Loulergue, and Masato Takeichi. Program calculation in Coq. In *Algebraic Methodology and Software Technology (AMAST)*. Springer-Verlag, Berlin, Heidelberg, 2011.

Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages as libraries. In *Programming Language Design and Implementation (PLDI)*. ACM, New York, NY, USA, 2011.

David Van Horn and Matthew Might. Abstracting abstract machines. In *International Conference on Functional Programming (ICFP)*. ACM, New York, NY, USA, 2010.

David Van Horn and Matthew Might. Systematic abstraction of abstract machines. In *Journal of Functional Programming (JFP)*. Cambridge University Press, Cambridge, UK, 2012.

Alexander Vandenbroucke, Tom Schrijvers, and Frank Piessens. Fixing nondeterminism. In *Implementation and Application of Functional Programming Languages (IFL)*. ACM, New York, NY, USA, 2015.

Dimitrios Vardoulakis. *CFA2: Pushdown Flow Analysis for Higher-Order Languages*. PhD thesis, Northeastern University, 2012.

Dimitrios Vardoulakis and Olin Shivers. CFA2: A context-free approach to control-flow analysis. In *European Symposium on Programming (ESOP)*. Springer-Verlag, Berlin, Heidelberg, 2010.

Dimitrios Vardoulakis and Olin Shivers. CFA2: a context-free approach to control-flow analysis. In *Logical Methods in Computer Science (LMCS)*. Logical Methods in Computer Science e.V., Braunschweig, Germany, 2011.

Andrew K. Wright and Suresh Jagannathan. Polymorphic splitting: An effective polyvariant flow analysis. In *Transactions on Programming Languages and Systems (TOPLAS)*. ACM, New York, NY, USA, 1998.

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Programming Language Design and Implementation (PLDI)*. ACM, New York, NY, USA, 2011.

Michael Zhivich and Robert K. Cunningham. The real cost of software errors. In *IEEE Security and Privacy*. IEEE, Washington D.C., USA, 2009.