Constructive Galois Connections

Taming the Galois Connection Framework for Mechanized Metatheory

David Darais
University of Maryland, USA
darais@cs.umd.edu

David Van Horn
University of Maryland, USA
dvanhorn@cs.umd.edu

Abstract

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.

This paper presents *constructive Galois connections*, a variant of Galois connections that is effective both on paper and in proof assistants; is complete with respect to 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 our 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 our metatheory.

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

Categories and Subject Descriptors F.3.2 [*Semantics of Programming Languages*]: Program analysis

Keywords Abstract Interpretation, Galois Connections, Monads

1. Introduction

Abstract interpretation is a general theory of sound approximation widely applied in programming language semantics, formal verification, and static analysis [10–14]. 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 many static analyzers, type systems, model-checkers, obfuscators, program transformations, and many more applications [7].

Given the remarkable set of intellectual 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 the goal of mechanization for the theory of abstract interpretation with Galois connections in Coq [26]. However, he notes that the abstraction side (α) of Galois connections poses a serious problem 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 generally be extracted.

Pichardie subsequently tackled the extraction problem by mechanizing a restricted formulation of abstract interpretation that relied only on the concretization (γ) side of Galois connections [29]. Doing so avoids the use of axioms and enables extraction of certified artifacts. This proof technique is effective and has been used to construct several certified static analyzers [1, 5, 6, 29], most notably the Verasco static analyzer, part of the CompCert C compiler [18, 19]. 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 [29, 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 [8]. It seems interesting to find a framework for this kind of symbolic manipulation, while remaining easily formalizable in Coq.

This important property is the so-called "calculational" style, whereby 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 style of abstract interpretation is detailed in Cousot's monograph [8], 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.

[1] Translated from French by the present authors.

ICFP'16, September 18–24, 2016, Nara, Japan
ACM. 978-1-4503-4219-3/16/09...$15.00
http://dx.doi.org/10.1145/2951913.2951934

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.

This paper contributes *constructive Galois connections*, a framework for mechanized 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. 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.

To support the utility of our theory we build a library for constructive Galois connections in Agda [28] and mechanize two existing abstract interpretation proofs from the literature. The first is drawn from Cousot's monograph [8], which derives a correct-by-construction analyzer from a specification induced by a concrete interpreter and Galois connection. The second is drawn from Garcia, Clark and Tanter's "Abstracting Gradual Typing" [17], 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, 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 itself mechanized; claims are marked with "AGDA✓" whenever they are proved in Agda. (All claims are marked.)

Contributions This paper 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.

The remainder of the paper is organized as follows. First we give a tutorial on verifying a simple analyzer from two different perspectives: direct verification (§2.1) and abstract interpretation with Galois connections (§2.2), highlighting mechanization issues along the way. We then present constructive Galois connections as a marriage of the two approaches (§3). We provide two case studies: the mechanization of an abstract interpreter from Cousot's calculational monograph (§4), and the mechanization of Garcia, Clark and Tanter's work on gradual typing *via* abstract interpretation (§5). Finally, we formalize the metatheory of constructive Galois connections (§6), relate our work to the literature (§7), and conclude (§8).

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 3 we present an alternative perspective: abstract interpretation with *constructive* Galois connections—the topic of this paper. Constructive Galois connections marry the 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.

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 $succ(n)$:

$$\mathbb{P} \coloneqq \{\texttt{EVEN}, \texttt{ODD}\} \qquad succ^\sharp(\texttt{EVEN}) \coloneqq \texttt{ODD}$$
$$succ^\sharp : \mathbb{P} \to \mathbb{P} \qquad succ^\sharp(\texttt{ODD}) \coloneqq \texttt{EVEN}$$

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

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

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 $p = \texttt{EVEN}$:

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

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 help foreshadow the constructive Galois connection setup.

$$parity : \mathbb{N} \to \mathbb{P} \qquad \begin{array}{l} parity(0) \coloneqq \texttt{EVEN} \\ parity(succ(n)) \coloneqq flip(parity(n)) \end{array}$$

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:

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

The soundness of the analyzer is then restated:

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

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

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

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 analysis results ($parity(n)$)—are central to the constructive Galois Connection framework we will describe in Section 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, propositions $P : \wp(A)$ are encoded constructively as undecidable predicates $P : A \to prop$ where $x \in P \Leftrightarrow P(x)$.

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

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

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 of the design are postulated, and which parts are derived. Using the direct approach, the analysis $succ^\sharp$, the interpretation for parities $[\![p]\!]$ and the definition of soundness are all 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 the soundness setup and the analyzer are both 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.

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 not designed, it is derived from the definition of abstraction. Finally, there is added boilerplate in the abstract interpretation approach, which requires lifting definitions and proofs to powersets $\wp(\mathbb{N})$ and $\wp(\mathbb{P})$.

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, at which point 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 $parity(n)$:

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

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

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

The correctness criteria for α and γ is the correspondence:

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

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

$$N \subseteq \gamma(\alpha(N)) \qquad \text{(GC-Exp)}$$
$$\alpha(\gamma(P)) \subseteq P \qquad \text{(GC-Red)}$$

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 also omitting monotonicity requirements for α and γ in our presentation (although (GC-Corr) implies monotonicity).

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:

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

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 $\uparrow succ$ (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 gives the researcher the choice between four soundness properties, all of which are equivalent and generated by α and γ:

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

Because each soundness property is equivalent (also a consequence of (GC-Corr)), 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 $\uparrow succ^\sharp$, a kind of completeness property, 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 $\uparrow succ^{\sharp\prime}$:

$$\uparrow succ^{\sharp\prime} : \wp(\mathbb{P}) \to \wp(\mathbb{P}) \qquad \uparrow succ^{\sharp\prime}(P) \coloneqq \{\texttt{EVEN}, \texttt{ODD}\}$$

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.

Just like soundness, four completeness statements are generated by α and γ, however each of the 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 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 a function. 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 $P = \{\texttt{EVEN}\}$:

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

The derivation of the other cases is 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 abstract interpretation framework, which shows one abandons simpler proof techniques in exchange for the benefits of abstract interpretation.

Resistance to Mechanized Verification Despite the beauty and utility of 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 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 static analyzers calls for constructive mathematics, a problem for α functions because they are often non-constructive, an observation first made by Monniaux [26]. To work around this limitation, Pichardie [29] advocates for designing abstract interpreters which are merely inspired by Galois connections, but ultimately avoiding their use in verification, which he terms the "γ-only" approach. Successful verification projects such as Verasco adopt this "γ-only" approach [18, 19], despite the use of Galois connections in designing the original Astrée analyzer [4].

To better understand the foundational issues with Galois connections and α functions, consider verifying the abstract interpretation approach to soundness for our parity analyzer using a proof assistant built on constructive logic. In this setting, the encoding of the Galois connection must support elements of infi-

nite 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 modelled constructively as a predicate $\mathbb{N} \to prop$. To support defining executable analyzers that manipulate sets of parities, the powerset $\wp(\mathbb{P})$ is modelled 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 $\uparrow succ^\sharp : \mathbb{P}^c \to \mathbb{P}^c$ which can be extracted and executed. The consequence of this design is a Galois connection between $\mathbb{N} \to prop$ and $\mathbb{P}^c$; the issue is now α:

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

This version of α cannot be defined constructively, as doing so requires deciding predicates over $\phi : \mathbb{N} \to prop$. To define α one must perform case analysis on predicates like $\exists n, \phi(n) \wedge even(n)$ to *compute* an element of $\mathbb{P}^c$, which is not possible for arbitrary ϕ. However, γ *can* be defined constructively:

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

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 [8]. 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.

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 mechanization.

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 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 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 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:

$$sound : x \in \mu(\eta(x)) \qquad \text{(CGC-Ext)}$$

$$tight : x \in \mu(y) \implies \eta(x) = y \qquad \text{(CGC-Red)}$$

Aside We use the term *extraction function* and notation η from Nielson *et al* [27] 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:

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

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

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

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

In the direct approach to verifying an example parity analysis described in Section 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 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 6. The case studies in Sections 4 and 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 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 $p = \texttt{EVEN}$:

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

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 2.1 and 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:

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

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 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.

3.1 Partial Orders and Monotonicity

The full theory of constructive Galois connections generalizes to posets $\langle A, \sqsubseteq \rangle^A$ 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 $A \searrow prop$, 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 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:

$$max^\sharp : \mathbb{P} \times \mathbb{P} \to \mathbb{P}$$
$$max^\sharp(\texttt{EVEN}, \texttt{EVEN}) \coloneqq \texttt{EVEN} \qquad max^\sharp(\texttt{EVEN}, \texttt{ODD}) \coloneqq ?$$
$$max^\sharp(\texttt{ODD}, \texttt{ODD}) \coloneqq \texttt{ODD} \qquad max^\sharp(\texttt{ODD}, \texttt{EVEN}) \coloneqq ?$$

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 $\texttt{ANY}$ to the set of parities: $\mathbb{P}^+ \coloneqq \mathbb{P} \cup \{\texttt{ANY}\}$; the new element $\texttt{ANY}$ is interpreted: $[\![\texttt{ANY}]\!] \coloneqq \{n \mid n \in \mathbb{N}\}$; the partial order on $\mathbb{P}^+$ becomes: $\texttt{EVEN}, \texttt{ODD} \sqsubseteq \texttt{ANY}$; and the correspondence continues to hold using this partial order: $n \in [\![p^+]\!] \iff parity(n) \sqsubseteq p^+$. $max^\sharp$ is then defined using the abstraction $\mathbb{P}^+$ and proven sound and optimal following the abstract interpretation paradigm.

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 6.

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

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

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 and 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 a constructive Galois is the Independent Attributes (IA) abstraction, which abstracts relations $R : \wp(A \times B)$ with their component-wise splitting $\langle R_l, R_r \rangle : \wp(A) \times \wp(B)$:

$$\alpha : \wp(A \times B) \to \wp(A) \times \wp(B)$$
$$\alpha(R) \coloneqq \langle \{x \mid \exists y. \langle x, y \rangle \in R\}, \{y \mid \exists x. \langle x, y \rangle \in R\} \rangle$$
$$\gamma : \wp(A) \times \wp(B) \to \wp(A \times B)$$
$$\gamma(R_l, R_r) \coloneqq \{\langle x, y \rangle \mid x \in R_l, y \in R_r\}$$

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

$$\alpha(R) \coloneqq \langle \lambda(x).\exists(y).R(x,y), \lambda(y).\exists(x).R(x,y) \rangle$$
$$\gamma(R_l, R_r) \coloneqq \lambda(x,y).R_l(x) \wedge R_r(y)$$

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.

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 6.

The monadic structure of classical powersets is standard, and is analogous to the nondeterminism monad familiar to Haskell programmers. However, the model $\wp(A) = A \to prop$ 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 3.1. This results in monotonic versions of monad operators ret and $bind$:

$$ret : A \nearrow \wp(A) \qquad bind : \wp(A) \times (A \nearrow \wp(B)) \nearrow \wp(B)$$
$$ret(x) \coloneqq \{x' \mid x' \sqsubseteq x\} \qquad bind(X, f) \coloneqq \{y \mid x \in X \wedge y \in f(x)\}$$

We adopt Moggi's notation [25] for monadic extension where $bind(X, f)$ 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 $A \to prop$ 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:

$$pure : (A \nearrow B) \nearrow (A \nearrow \wp(B)) \quad pure(f)(x) \coloneqq ret(f(x))$$

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

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

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

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

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:

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

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

Figure 1. Case Study: `WHILE` abstract syntax

4. Case Study: Calculational AI

In this section we apply constructive Galois connections to the *Calculational Design of a Generic Abstract Interpreter* from Cousot's monograph [8]. 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 the abstraction mapping without the use of axioms, and the 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 3 and formalize in Section 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.

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 1. `WHILE` syntactically distinguished arithmetic, boolean and command expressions. `rand` is an arithmetic expression which can evaluate to any integer. Syntactic categories $\oplus$, $\ominus$ and $\otimes$ range over arithmetic, comparison and boolean operators, and are introduced to simplify the presentation. The `WHILE` language is taken from Cousot's monograph [8].

The concrete semantics of `WHILE` is sketched without full definition in Figure 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 expressions and commands due to the nondeterminism of `rand` and nontermination of `while`. These semantics serve as the starting point for designing an abstract interpreter.

$$
\begin{array}{rr}
\rho \in \texttt{env} := \texttt{var} \rightharpoonup \mathbb{Z} & \varsigma \in \Sigma ::= \langle \rho, ce \rangle \\
[\![\_]\!]^a \in aop \to \mathbb{Z} \times \mathbb{Z} \rightharpoonup \mathbb{Z} & \_ \vdash \_ \Downarrow^a \_ \in \wp(\texttt{env} \times \texttt{aexp} \times \mathbb{Z}) \\
[\![\_]\!]^c \in cmp \to \mathbb{Z} \times \mathbb{Z} \to \mathbb{B} & \_ \vdash \_ \Downarrow^b \_ \in \wp(\texttt{env} \times \texttt{bexp} \times \mathbb{B}) \\
[\![\_]\!]^b \in bop \to \mathbb{B} \times \mathbb{B} \to \mathbb{B} & \_ \mapsto^c \_ \in \wp(\Sigma \times \Sigma)
\end{array}
$$

$$\frac{}{\rho \vdash \texttt{rand} \Downarrow^a i}\ \textsc{ARand}$$

$$\frac{\rho \vdash ae_1 \Downarrow^a i_1 \qquad \rho \vdash ae_2 \Downarrow^a i_2}{\rho \vdash ae_1 \oplus ae_2 \Downarrow^a [\![\oplus]\!]^a(i_1, i_2)}\ \textsc{AOp}$$

$$\frac{\rho \vdash ae \Downarrow^a i}{\langle \rho, x := ae \rangle \mapsto^c \langle \rho[x \leftarrow i], \texttt{skip} \rangle}\ \textsc{CAssign}$$

$$\frac{\rho \vdash be \Downarrow^b true}{\langle \rho, \texttt{while}\ be\ \texttt{do}\ ce \rangle \mapsto^c \langle \rho, ce\ ;\ \texttt{while}\ be\ \texttt{do}\ ce \rangle}\ \textsc{CWhile-T}$$

$$\frac{\rho \vdash be \Downarrow^b false}{\langle \rho, \texttt{while}\ be\ \texttt{do}\ ce \rangle \mapsto^c \langle \rho, \texttt{skip} \rangle}\ \textsc{CWhile-F}$$

Figure 2. Case Study: `WHILE` concrete semantics

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 paper for the full mechanization of `WHILE`.

Abstracting Integers We design a simple sign abstraction for integers, although more powerful abstractions are certainly possible [24]. 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:

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

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

$$
\begin{array}{ll}
\mu^z : \mathbb{Z}^\sharp \nearrow \wp(\mathbb{Z}) & \\
\mu^z(\texttt{none}) := \{\} & \mu^z(\texttt{negz}) := \{i \mid i \leq 0\} \\
\mu^z(\texttt{neg}) := \{i \mid i < 0\} & \mu^z(\texttt{nzer}) := \{i \mid i \neq 0\} \\
\mu^z(\texttt{zer}) := \{0\} & \mu^z(\texttt{posz}) := \{i \mid i \geq 0\} \\
\mu^z(\texttt{pos}) := \{i \mid i > 0\} & \mu^z(\texttt{any}) := \{i \mid i \in \mathbb{Z}\}
\end{array}
$$

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

$$\texttt{neg} \sqsubseteq^z \texttt{negz}, \texttt{nzer} \qquad \texttt{pos} \sqsubseteq^z \texttt{nzer}, \texttt{posz}$$
$$\texttt{zer} \sqsubseteq^z \texttt{negz}, \texttt{posz} \qquad \texttt{none} \sqsubseteq^z i^\sharp \sqsubseteq^z i^\sharp \sqsubseteq^z \texttt{any}$$

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

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

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:

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

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.

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

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

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

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 obeys $\rho_1^\sharp \sqsubseteq^r \rho_2^\sharp \iff \mu^r(\rho_1^\sharp) \subseteq \mu^r(\rho_2^\sharp)$:

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

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

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

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:

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

which is also not amenable to mechanized verification.

Abstracting Functions After designing constructive Galois connections for $\mathbb{Z}$ and env we define what it means for $[\![\_]\!]^{\sharp a}$, 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 $[\![\_]\!]^{\sharp a}$ is generated using the constructive Galois connection for $\mathbb{Z}$:

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

(See (CGC-Cmp/$\eta\mu$) in Section 3 for the origin of this equation.) For $[\![\_]\!]^{\sharp a}$, we postulate its definition and verify its correctness post-facto using the above property, although we omit the proof details here. The definition of $[\![\_]\!]^{\sharp a}$ is standard, and returns none in the case of division by zero. We show only the definition of $+$ here:

$$[\![\_]\!]^{\sharp a} : \texttt{aexp} \to \mathbb{Z}^\sharp \times \mathbb{Z}^\sharp \nearrow \mathbb{Z}^\sharp$$
$$[\![+]\!]^{\sharp a}(i_1^\sharp, i_2^\sharp) \coloneqq \bigsqcup \begin{cases} \texttt{pos} & \textit{if } \texttt{pos} \sqsubseteq^z i_1^\sharp \vee \texttt{pos} \sqsubseteq^z i_2^\sharp \\ \texttt{neg} & \textit{if } \texttt{neg} \sqsubseteq^z i_1^\sharp \vee \texttt{neg} \sqsubseteq^z i_2^\sharp \\ \texttt{zer} & \textit{if } \texttt{zer} \sqsubseteq^z i_1^\sharp \wedge \texttt{zer} \sqsubseteq^z i_2^\sharp \\ \texttt{zer} & \textit{if } \texttt{pos} \sqsubseteq^z i_1^\sharp \wedge \texttt{neg} \sqsubseteq^z i_2^\sharp \\ \texttt{zer} & \textit{if } \texttt{neg} \sqsubseteq^z i_1^\sharp \wedge \texttt{pos} \sqsubseteq^z i_2^\sharp \end{cases}$$

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:

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

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

$$\alpha^z([\![ae]\!]^a_\wp(\gamma^z(i_1^\sharp, i_2^\sharp))) = [\![ae]\!]^{\sharp a}(i_1^\sharp, i_2^\sharp)$$

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 $[\![\_]\!]^{\sharp a}$ using constructive Galois connections. The state-of-the art approach of "γ-only" verification would instead mechanize the $\gamma\gamma$ variant of correctness:

$$[\![ae]\!]^a_\wp(\gamma^z(i_1^\sharp, i_2^\sharp)) = \gamma^z([\![ae]\!]^{\sharp a}(i_1^\sharp, i_2^\sharp))$$

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

$$\langle i_1, i_2\rangle \in \mu^{\tilde{z}}(i_1^\sharp, i_2^\sharp) \wedge \langle i_1', i_2'\rangle = [\![ae]\!]^a(i_1, i_2) \Leftrightarrow \langle i_1', i_2'\rangle \in \mu^{\tilde{z}}([\![ae]\!]^{\sharp a}(i_1^\sharp, i_2^\sharp))$$

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. $\mathcal{A}^\sharp$ has type:

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

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

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

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

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

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 3. Constructive GC calculations on paper

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 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. 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 ⟅e⟆ terms, which transform the proof state from the previous form to the next. The term $[\texttt{focus-right}\ [\cdot]\ of\ e]$ 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.

We complete the full calculation of Cousot's generic abstract interpreter for `WHILE` in Agda as supplemental material to this paper, 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.

```
-- Agda Calculation of Case ae = x:
α[A] (Var x) ρ♯ = [proof-mode]
  do [[ (pure · η^z) ∗· (A[ Var x ] ∗· (μ^r · ρ♯)) ]]
   . [focus-right [·] of (pure · η^z) ∗ ] begin
    do [[ A[ Var x ] ∗· (μ^r · ρ♯) ]]
     . ⟅ A[Var]/≡ ⟆
     . [[ (pure · lookup[ x ]) ∗· (μ^r · ρ♯) ]]
     . ⟅ lookup/μ^r/≡ ⟆
     . [[ μ^z ∗· (pure · lookup♯[ x ] · ρ♯) ]]
    end
   . [[ (pure · η^z) ∗· (μ^z ∗· (pure · lookup♯[ x ] · ρ♯)) ]]
   . ⟅ reductive[ημ] ⟆
   . [[ ret · (lookup♯[ x ] · ρ♯) ]]
   . [[ pure · A♯[ Num n ] · ρ♯ ]] □
```

Figure 4. Constructive GC calculations in Agda

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

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

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.

5. Case Study: Gradual Type Systems

Recent work in metatheory for gradual type systems by Garcia et al. [17] 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 in the lattice of type precision. The precise typing rules are presented with meta-operators $<:$ for subtyping and $\ddot{\vee}$ for the join operator in the subtyping lattice. The gradual type system is then written using abstract variants $<:^\sharp$ 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

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \tau_1 <: \mathbb{B} \quad \Gamma \vdash e_2 : \tau_2 \quad \Gamma \vdash e_3 : \tau_3}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau_1 \ddot{\vee} \tau_2}\ \textsc{If}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \tau_1 <: \tau_{11} \to \tau_{21} \quad \Gamma \vdash e_2 : \tau_2 \qquad \tau_2 <: \tau_{11}}{\Gamma \vdash e_1(e_2) : \tau_{21}}\ \textsc{App}$$

$$\frac{\Gamma \vdash e : \tau_1 \qquad \tau_1 <: \tau_2}{\Gamma \vdash e :: \tau_2 : \tau_2}\ \textsc{Coe}$$

Figure 5. Case Study: precise type system

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 5. The subtyping lattice in the 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 is reflexive and has ? at the top:

$$\tau^\sharp \sqsubseteq \tau^\sharp \sqsubseteq ?$$

And arrow types are monotonic:

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

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:

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

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 $\ddot{\vee}^\sharp$:

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

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

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

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

$$\frac{\Gamma^\sharp \vdash^\sharp e_1^\sharp : \tau_1^\sharp \qquad \tau_1^\sharp <:^\sharp \mathbb{B} \quad \Gamma^\sharp \vdash^\sharp e_2^\sharp : \tau_2^\sharp \quad \Gamma^\sharp \vdash^\sharp e_3^\sharp : \tau_3^\sharp}{\Gamma^\sharp \vdash^\sharp \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau_1^\sharp \ddot{\vee}^\sharp \tau_2^\sharp}\ \textsc{G-If}$$

$$\frac{\Gamma^\sharp \vdash^\sharp e_1^\sharp : \tau_1^\sharp \qquad \tau_1^\sharp <:^\sharp \tau_{11}^\sharp \to \tau_{21}^\sharp \quad \Gamma^\sharp \vdash^\sharp e_2^\sharp : \tau_2^\sharp \qquad \tau_2^\sharp <:^\sharp \tau_{11}^\sharp}{\Gamma^\sharp \vdash e_1^\sharp(e_2^\sharp) : \tau_{21}^\sharp}\ \textsc{G-App}$$

$$\frac{\Gamma^\sharp \vdash^\sharp e^\sharp : \tau_1^\sharp \qquad \tau_1^\sharp <:^\sharp \tau_2^\sharp}{\Gamma^\sharp \vdash^\sharp e^\sharp :: \tau_2^\sharp : \tau_2^\sharp}\ \textsc{G-Coe}$$

Figure 6. Case Study: 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 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 gradual guarantee (GG):

$$\vdash e : \tau \iff \vdash^\sharp e : \tau \qquad \text{(FAT)}$$

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

$$\vdash^\sharp e_1^\sharp : \tau_1^\sharp \wedge e_1^\sharp \sqsubseteq e_2^\sharp \implies \vdash^\sharp e_2^\sharp : \tau_2^\sharp \wedge \tau_1^\sharp \sqsubseteq \tau_2^\sharp \qquad \text{(GG)}$$

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 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 using the constructive theorem of choice, as depicted in Figure 8.

Classical Galois Connections We review classical Galois connections in Figure 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 3.1 and 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 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

Adjunction	Classical GCs	Kleisli/constructive GCs
Category	Posets	Posets
Adjoints	Mono. Functions	$\wp$-Monadic Functions
LAdjoint	$\alpha : A \nearrow B$	$\kappa\alpha : A \nearrow \wp(B)$
RAdjoint	$\gamma : B \nearrow A$	$\kappa\gamma : B \nearrow \wp(A)$
Corr	$id(x) \sqsubseteq \gamma(y)$ $\Leftrightarrow \alpha(x) \sqsubseteq id(y)$	$ret(x) \subseteq \kappa\gamma(y)$ $\Leftrightarrow \kappa\alpha(x) \subseteq ret(y)$
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 7. Comparison of constructive v classical adjunctions

lifting operator on Kleisli Galois connections $\langle \kappa\alpha, \kappa\gamma \rangle^*$:

$$\langle \alpha, \gamma \rangle = \langle \kappa\alpha, \kappa\gamma \rangle^* = \langle \kappa\alpha^*, \kappa\gamma^* \rangle$$

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

Theorem 1 (KGC-Sound)**.** *AGDA✓ 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.*

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)**.** *AGDA✓ 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.*

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 $\langle \kappa\alpha, \kappa\gamma \rangle = \langle pure(\eta), \mu \rangle$. Soundness and optimality theorems are then lifted from constructive to Kleisli without modification.

Theorem 3 (CGC-Sound)**.** *AGDA✓ 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 classical 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.*

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

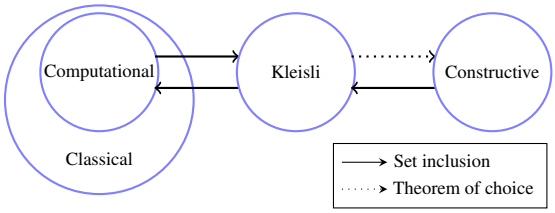

Figure 8. Relationship between classical, Kleisli and constructive

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

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.

Proof. To turn an arbitrary Kleisli Galois connection into a constructive one, we show that the effect on $\kappa\alpha : A \nearrow \wp(B)$ is benign, or in other words, that there exists some η such that $\kappa\alpha = pure(\eta)$. 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:

```
choice : ∀{A B}{R : A → B → Set}
→ (∀ x → ∃ y st R x y)
→ (∃ f st ∀ x → R x (f x))
choice P = ∃ (λ x → π₁ (P x)) , (λ x → π₂ (P x))
```

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

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

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

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

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

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

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

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

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)**.** *AGDA✓ 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)**.** *AGDA✓ 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.*

Mechanization We mechanize the metatheory for constructive Galois connections and both case studies from Sections 4 and 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, 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.

7. 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 [11, 12]; the latter on that of Martin-Löf [21], embodied in Norell's Agda [28]. Our key technical insight is to use a monadic structure for Galois connections, following the example of Moggi [25] for the λ-calculus.

Calculational Abstract Interpretation Cousot describes calculational abstract interpretation by example in his lecture notes [9] and monograph [8], and recently introduced a unifying calculus for Galois connections [14]. Our work mechanizes Cousot's calculations and provides a foundation for mechanizing other instances of calculational abstract interpretation (e.g. [22, 30]). We expect our work to have applications to the mechanization of calculational program design [2, 3] by employing only Galois *retractions*, i.e. $\alpha \circ \gamma$ is an identity [14]. There is prior work on mechanized program calculation [33], but it is not based on abstract interpretation.

Verified Static Analyzers Verified abstract interpretation has many promising results [1, 5, 6, 29], scaling up to large-scale real-world static analyzers [18]. 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* [32] 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 [16] 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 justified explicitly.

Monadic Abstract Interpretation Monads in abstract interpretation have recently been applied to good effect for modularity [15, 31]. 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. [23, 30]). 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[16]. 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 [20]) or Agda to automate these proofs in a way that maintains proof-checker performance.

There have been recent developments on compositional abstract interpretation frameworks [15] where abstract interpreters and their proofs of soundness are systematically derived side-by-side. That framework relies on correctness properties transported by *Galois transformers*, which we posit would benefit from mechanization since they hold both computational and specification content.

8. Conclusions

This paper realizes the vision of mechanized and constructive Galois connections foreshadowed by Cousot [8, 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]

Acknowledgments

We thank Ron Garcia and Éric Tanter for discussions of their work. Éric also helped with our French translation. We thank the Colony Club in D.C. and the Board & Brew in College Park for providing fruitful environments in which to work. We thank the anonymous reviewers of ICFP 2016 for their helpful feedback. This material is partially based on research sponsored by DARPA under agreement number AFRL FA8750-15-2-0104.

[2] `di.ens.fr/~cousot/aisoftware/Marktoberdorf98/Bug_History`

References

[1] G. Barthe, D. Pichardie, and T. Rezk. A certified lightweight non-interference java bytecode verifier. In R. D. Nicola, editor, *Programming Languages and Systems*, Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007.

[2] R. Bird and O. de Moor. *The Algebra of Programming*. Prentice Hall, 1996.

[3] R. S. Bird. A calculus of functions for program derivation. In D. A. Turner, editor, *Research Topics in Functional Programming*. Addison-Wesley, 1990. Also available as Technical Monograph PRG-64, from the Programming Research Group, Oxford University.

[4] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '03. ACM, 2003.

[5] S. Blazy, V. Laporte, A. Maroneze, and D. Pichardie. Formal verification of a c value analysis based on abstract interpretation. In F. Logozzo and M. Fähndrich, editors, *Static Analysis*, Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013.

[6] D. Cachera and D. Pichardie. A certified denotational abstract interpreter. In M. Kaufmann and L. Paulson, editors, *Interactive Theorem Proving*, Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010.

[7] P. Cousot. Abstract interpretation. URL `http://www.di.ens.fr/~cousot/AI/`.

[8] P. Cousot. The calculational design of a generic abstract interpreter. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design*. NATO ASI Series F. IOS Press, Amsterdam, 1999.

[9] P. Cousot. MIT 16.399: Abstract interpretation, 2005.

[10] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *2nd International Symposium on Programming*, 1976.

[11] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. ACM, 1977.

[12] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '79. ACM, 1979.

[13] P. Cousot and R. Cousot. Inductive definitions, semantics and abstract interpretations. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '92. ACM, 1992.

[14] P. Cousot and R. Cousot. A Galois connection calculus for abstract interpretation. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14. ACM, 2014.

[15] D. Darais, M. Might, and D. Van Horn. Galois transformers and modular abstract interpreters. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 552–571. ACM, 2015.

[16] B. Delaware, C. Pit-Claudel, J. Gross, and A. Chlipala. Fiat: Deductive synthesis of abstract data types in a proof assistant. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2015. ACM, 2015.

[17] R. Garcia, A. M. Clark, and É. Tanter. Abstracting gradual typing. In *Proceedings of the 43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2016)*, St Petersburg, FL, USA, Jan. 2016. ACM Press. To appear.

[18] J. H. Jourdan, V. Laporte, S. Blazy, X. Leroy, and D. Pichardie. A Formally-Verified c static analyzer. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15. ACM, 2015.

[19] X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 2009.

[20] G. Malecha and J. Bengtson. Extensible and efficient automation through reflective tactics. In *Programming Languages and Systems - 25th European Symposium on Programming*, ESOP 2016. Springer, 2016.

[21] P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984.

[22] J. Midtgaard and T. Jensen. A calculational approach to Control-Flow analysis by abstract interpretation. In M. Alpuente and G. Vidal, editors, *Static Analysis*, Lecture Notes in Computer Science, chapter 23. Springer Berlin Heidelberg, 2008.

[23] J. Midtgaard and T. Jensen. A calculational approach to Control-Flow analysis by abstract interpretation. In M. Alpuente and G. Vidal, editors, *Static Analysis Symposium*, LNCS. Springer, 2008.

[24] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 2006.

[25] E. Moggi. An abstract view of programming languages. Technical report, Edinburgh University, 1989.

[26] D. Monniaux. Réalisation mécanisée d'interpréteurs abstraits. Rapport de DEA, Université Paris VII, 1998. French.

[27] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[28] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, Sept. 2007.

[29] D. Pichardie. *Interprétation abstraite en logique intuitionniste: extraction d'analyseurs Java certifiés*. PhD thesis, Université Rennes, 2005.

[30] I. Sergey, J. Midtgaard, and D. Clarke. Calculating graph algorithms for dominance and shortest path. In J. Gibbons and P. Nogueira, editors, *Mathematics of Program Construction*, Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012.

[31] I. Sergey, D. Devriese, M. Might, J. Midtgaard, D. Darais, D. Clarke, and F. Piessens. Monadic abstract interpreters. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2013.

[32] P. F. Silva and J. N. Oliveira. 'Galculator': Functional prototype of a Galois-connection based proof assistant. In *Proceedings of the 10th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, PPDP '08. ACM, 2008.

[33] J. Tesson, H. Hashimoto, Z. Hu, F. Loulergue, and M. Takeichi. Program calculation in Coq. In M. Johnson and D. Pavlovic, editors, *Algebraic Methodology and Software Technology*, Lecture Notes in Computer Science, chapter 10. Springer Berlin Heidelberg, 2011.