Mechanically Verified Calculational Abstract Interpretation

David Darais
University of Maryland
davdar@cs.umd.edu

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

Abstract

Calculational abstract interpretation, long advocated by Cousot, is a technique for deriving correct-by-construction abstract interpreters from the formal semantics of programming languages.

This paper addresses the problem of deriving correct-by-verified-construction abstract interpreters with the use of a proof assistant. We identify several technical challenges to overcome with the aim of supporting verified calculational abstract interpretation that is faithful to existing pencil-and-paper proofs, supports calculation with Galois connections generally, and enables the extraction of verified static analyzers from these proofs.

To meet these challenges, we develop a theory of Galois connections in monadic style that include a specification effect. Effectful calculations may reason classically, while pure calculations have extractable computational content. Moving between the worlds of specification and implementation is enabled by our metatheory.

To validate our approach, we give the first mechanically verified proof of correctness for Cousot’s “Calculational design of a generic abstract interpreter.” Our proof “by calculus” closely follows the original paper-and-pencil proof and supports the extraction of a verified static analyzer.

Keywords Abstract interpretation, Galois connections, dependently typed programming, mechanized metatheory, static analysis

1. Introduction

Abstract interpretation [9, 10] is a foundational and unifying theory of semantics and abstraction developed by P. Cousot and R. Cousot, which has had notable impact on the theory and practice of program analysis and verification. Traditionally, static analyses and verification frameworks such as type systems, program logics, or constraint-based analyses start by first postulating a specification of an abstract semantics. Only afterward is this abstraction proved correct with respect to the language’s semantics. This proof establishes post facto that the analysis or logic is an abstract interpretation of the underlying language semantics.

P. Cousot has also advocated an alternative approach to the design of abstract interpreters called calculational abstract interpretation [7, 8], which involves systematically applying abstraction functions to a programming language semantics in order to derive an abstraction. Abstract interpretations derived in the calculational style are correct by construction (assuming no missteps are made in the calculation) and need not be proved sound after the fact.

This paper addresses the problem of mechanically verifying the derivations of calculational abstract interpretation using a proof assistant. We identify several technical challenges to modelling the theory of abstract interpretation in a constructive, dependent type theory and then develop solutions to these challenges. Paramount in overcoming these challenges is effectively representing Galois connections and maintaining a modality between specifications and implementations to enable program extraction. To do this, we propose a novel form of Galois connections endowed with monadic structure which we dub Kleisli Galois connections. This monadic structure maintains a distinction between calculation at the specification level, which may be non-constructive, and at the implementation level, which must be constructive. Remarkably, calculations are able to move back and forth between these modalities and verified programs may be extracted from the end result of calculation.

To establish the adequacy of our theory, we prove it is sound and complete with respect to a subset of traditional Galois connections, and isomorphic to a space of fully constructive Galois connections, diagrammed in figure 1. To establish the utility of our theory, we construct a framework for abstract interpretation with Kleisli Galois connections in the dependently typed programming language and proof-assistant, Agda [20]. To validate our method, we re-derive Cousot’s generic compositional static analyzer for an imperative language by abstract interpretation of the language’s formal semantics. Consequently we obtain a verified proof of the calculation and extract a verified implementation of Cousot’s static analyzer.

Contributions This paper contributes:

1. a framework for mechanically verified abstract interpretation that supports calculation and program extraction,
2. a theory of specification effects in Galois connections, and
3. a verified proof of Cousot’s generic abstract interpreter derived by calculus.

To supplement these contributions, we provide two artifacts. The first is the source code of this document, which is a literate Agda program and verified at typesetting-time. For presentation purposes, it assumes a few lemmas and is less general than it could be. The second artifact is a stand-alone Agda program that develops all of the results in this paper in full detail, including the mathematically stated theorems and lemmas. Claims are marked with a ✓ whenever they have been proved in Agda. (All claims are checked.) The full development is found at:

https://github.com/plum-umd/mvcai

Although largely self-contained, this paper assumes a basic familiarity with abstract interpretation and dependently typed programming. There are excellent tutorials on both ([8, 11] and [5, 21], respectively).
2. Calculational Abstract Interpretation

To demonstrate our approach to mechanizing Galois connections we present the calculation of a generic abstract interpreter, as originally presented by Cousot [7]. The setup is a simple arithmetic expression language which includes a random number representation, and is otherwise standard. The syntax and semantics is given in figure 2.

A collecting semantics is defined as a monotononic (written \(\rightarrow\)) predicate transformer using \(\_ \vdash \_ \rightarrow \_\).

\[
\begin{align*}
eval &: \text{exp} \rightarrow \varphi(\text{env}) \rightarrow \varphi(\text{val}) \\
\eval[R] \vdash (v | \exists \rho \in R : \rho \vdash e \mapsto v)
\end{align*}
\]

In the setting of abstract interpretation, an analysis for a program \(e\) is performed by: (1) defining another semantics \(\eval^\sharp\), where \(\eval^\sharp\) is shown to soundly approximate the semantics of \(\eval\), and (2) executing the \(\eval^\sharp\) semantics and observing the output. There are many different methods for arriving at \(\eval^\sharp\), however the calculational approach prescribes a methodology for defining \(\eval^\sharp\) through calculus, the results of which are correct by construction.

To arrive at \(\eval^\sharp\) through calculus we first establish an abstraction for the domain \(\varphi(\text{env}) \rightarrow \varphi(\text{val})\), which we call \(\eval^\flat \rightarrow \eval^\sharp\). After abstracting the domain, we induce a best specification for any abstract semantics \(\eval^\sharp \in \eval^\flat \rightarrow \eval^\sharp\). Then we perform calculation on this specification to arrive at a definition for \(\eval^\sharp\). Key in this methodology is the requirement that \(\eval^\sharp\) be an algorithm, otherwise we would just define \(\eval^\sharp\) to be the induced best specification and be done.

We induce the best specification for \(\eval\) by: (1) constructing an abstraction for values \(\eval^\flat\) and proving it is a valid abstraction of \(\varphi(\text{val})\), (2) constructing an abstraction for environments \(\eval^\flat\) and proving it is a valid abstraction of \(\varphi(\text{env})\) in \(\varphi(\text{val})\), and (3) lifting these abstractions pointwise to \(\eval^\flat \rightarrow \eval^\sharp\) and proving it is a valid abstraction of \(\varphi(\text{env}) \rightarrow \varphi(\text{val})\), and (4) inducing \(\eval^\flat \rightarrow \eval^\flat\) as the best abstraction of \(\eval\) using the results from (3).

**Abstracting values** We pick a simple sign abstraction for \(\eval^\flat\), however our final abstract interpreter will be fully generic to \(\eval^\flat\), as is done in Cousot’s original derivations [7].

\[
\varphi^\sharp \in \varphi^\flat := \{-, 0, +, \top, \bot\}
\]

The set \(\varphi^\flat\) has the partial ordering \(\bot \subseteq - \subseteq 0 \subseteq + \subseteq \top\) where \(\_ \vdash \_\) is notation for incomparable.

Justifying that \(\varphi^\flat\) is a valid abstraction takes the form of a Galois connection:

\[
\varphi(\text{val}) \leftrightarrow \varphi^\flat \rightarrow \varphi(\text{val})^\flat.
\]

Galois connections are mappings between concrete objects and abstract objects which satisfy soundness and completeness properties. For \(\varphi^\flat\), the Galois connection with \(\varphi(\text{val})\) is defined:

\[
\begin{align*}
\alpha^\flat &: \varphi(\text{val}) \rightarrow \varphi^\flat \\
\gamma^\flat &: \varphi^\flat \rightarrow \varphi(\text{val}) \\
\alpha^\flat(V) &= \alpha^\flat(v) = \{v | v < 0\} \\
\gamma^\flat(0) &= \{0\} \\
\alpha^\flat(+V) &= \alpha^\flat(v) = \{v | v > 0\} \\
\gamma^\flat(\top) &= \mathbb{Z} \\
\gamma^\flat(\bot) &= \emptyset
\end{align*}
\]

\(\alpha^\flat\) is called the abstraction function, which maps concrete sets of numbers in \(\varphi(\text{val})\) to a finite, symbolic representations in \(\varphi^\flat\). \(\gamma^\flat\) is called the concretization function, mapping abstract symbols in \(\varphi^\flat\) concrete sets in \(\varphi(\text{val})\).

This Galois connection is extensive: properties of values in \(\varphi^\flat\) imply properties of related concrete values in \(\varphi(\text{val})\). It is reductive: \(\alpha^\flat\) is the best possible abstraction given \(\gamma^\flat\).

**Abstracting environments** We abstract \(\varphi(\text{env})\) with \(\varphi^\flat\):

\[
\begin{align*}
\rho &\vdash n \mapsto n \\
\rho &\vdash e \mapsto v
\end{align*}
\]

Justifying that \(\varphi^\flat\) is a valid abstraction is done through a Galois connection \(\varphi(\text{env}) \leftrightarrow \varphi^\flat\rightarrow \varphi(\text{val})^\flat\).

\[
\begin{align*}
\alpha^\flat &: \varphi(\text{env}) \rightarrow \varphi^\flat \\
\alpha^\flat(R) &= \mathbb{Z}(\lambda x.\alpha^\flat(\rho(x) | \rho \in R)) \\
\gamma^\flat &: \varphi^\flat \rightarrow \varphi(\text{env}) \\
\gamma^\flat(\rho^\flat) &= \{\rho | \forall (x.\rho(x) \in \gamma^\flat(\rho^\flat(x)))\}
\end{align*}
\]

Lemma 1 (extensive).\(\alpha^\flat \circ \gamma^\flat\) is extensive, that is:

\[
\forall (V \in \varphi(\text{val})).V \subseteq \gamma(\alpha^\flat(V)).
\]

Lemma 2 (reductive).\(\alpha^\flat \circ \gamma^\flat\) is reductive, that is:

\[
\forall (v^\flat \in \varphi^\flat).\alpha^\flat(\gamma^\flat(v^\flat)) \subseteq v^\flat.
\]

Abstracting the function space To abstract \(\varphi(\text{env}) \rightarrow \varphi(\text{val})\), we abstract its components pointwise with \(\varphi^\flat \rightarrow \varphi^\flat\), and justify the abstraction with another Galois connection.

\[
\begin{align*}
\alpha^\flat &: \varphi(\text{env}) \rightarrow \varphi(\text{val})^\flat \\
\gamma^\flat &: \varphi(\text{val}) \rightarrow \varphi(\text{val})^\flat \\
\gamma^\flat(\rho^\flat) &= \{\rho | \forall (x.\rho(x) \in \gamma^\flat(\rho^\flat(x)))\}
\end{align*}
\]

Lemma 3 (extensive).\(\alpha^\flat \circ \gamma^\flat\) is extensive, that is:

\[
\forall (R \in \varphi(\text{env})).R \subseteq \gamma(\alpha^\flat(R)).
\]

Lemma 4 (reductive).\(\alpha^\flat \circ \gamma^\flat\) is reductive, that is:

\[
\forall (\rho^\flat \in \varphi^\flat).\alpha^\flat(\gamma^\flat(\rho^\flat)) \subseteq \rho^\flat.
\]

Figure 2: Syntax and semantics

| symbol | description |
|--------|-------------|
| \(\alpha^\flat\) | abstraction function |
| \(\gamma^\flat\) | concretization function |
| \(\alpha^\flat(V)\) | \(\{v | v < 0\}\) |
| \(\gamma^\flat(0)\) | \(\{0\}\) |
| \(\alpha^\flat(+V)\) | \(\{v | v > 0\}\) |
| \(\gamma^\flat(\top)\) | \(\mathbb{Z}\) |
| \(\gamma^\flat(\bot)\) | \(\emptyset\) |

| symbol | description |
|--------|-------------|
| \(\rho \vdash n \mapsto n\) | Environment declaration |
| \(\rho \vdash e \mapsto v\) | Environment evaluation |

\n
\[\begin{array}{|c|c|c|}
\hline
n & \text{lit} & \mathbb{Z} \\
\hline
x & \text{var} & \text{variables} \\
\hline
u & \text{unary} & +, - \\
\hline
b & \text{binary} & +, |, \times, \% \\
\hline
e & \text{exp} & n \\
\hline
\end{array}\]

\[
\begin{align*}
\rho &\vdash u \mapsto \rho(\rho) \\
\rho &\vdash e \mapsto \rho(x) \\
\rho &\vdash \text{rand} \mapsto n
\end{align*}
\]

\[
\begin{align*}
\rho &\vdash e \mapsto v \\
\rho &\vdash e \mapsto x
\end{align*}
\]

\[
\begin{align*}
\rho &\vdash e \mapsto v \\
\rho &\vdash e \mapsto v
\end{align*}
\]
Numeric literals
\[ \alpha^v(\{v\mid \exists p \in \gamma \cdot v = \gamma(p^2) : \rho \vdash n \mapsto v\}) \]
\[ = \alpha^v([n] \rho^2) \]
\[ \pm \text{eval}^f([n] \rho^2) \]
definition of \(\rho \vdash n \mapsto v\)
\[ \text{by defining } \text{eval}^f([n] \rho^2) := \alpha^v([n]) \]

Variable Reference
\[ \alpha^v(\{v\mid \exists p \in \gamma \cdot v = \gamma(p^2) : \rho \vdash x \mapsto v\}) \]
\[ = \alpha^v((\lambda(\rho(x) \mid \rho \in \gamma(p^2)))) \]
\[ = \alpha^v((\lambda(\rho(x) \mid \rho \in \gamma(p^2)))) \]
\[ \subseteq \rho^2(x) \]
\[ \pm \text{eval}^f[x] \rho^2 \]
definition of \(\rho \vdash x \mapsto v\)

Unary operators
\[ \alpha^v(\{v\mid \exists p \in \gamma \cdot v = \gamma(p^2) : \rho \vdash u \mapsto v\}) \]
\[ = \alpha^v((\lambda(\rho(x) \mid \rho \in \gamma(p^2)))) \]
\[ \subseteq \alpha^v((\lambda(\rho(x) \mid \rho \in \gamma(p^2)))) \]
\[ \equiv \alpha^v((\lambda(\rho(x) \mid \rho \in \gamma(p^2)))) \]
\[ \subseteq \text{eval}^f \rho^2 \]
definition of \(\rho \vdash u \mapsto v\)

\text{eval}^f is an algorithm, at which point we have defined \text{eval}^f[e] through calculation.

2.1 Calculating the Abstract Interpreter

The calculation of \text{eval}^f begins by expanding definitions:

\[ c_{\text{eval}^f}(e) \]
\[ = v \]
\[ = \alpha^v((\text{eval}^f[e])(\gamma(p^2))) \]
\[ = \alpha^v((\text{eval}^f[e])(\gamma(p^2))) \]
\[ \pm \text{eval}^f \rho^2 \]
definition of \(c_{\text{eval}^f}(e)\)

In case \(\gamma(p^2) = \emptyset\), then have \(\alpha^v((\text{eval}^f[e])(\rho^2)) = \alpha^v(\emptyset) = \bot\). Otherwise, we proceed by induction on \(e\), assuming \(\gamma(p^2)\) is nonempty.

In figure 3, we show the calculations for literals, variables, and unary operator expressions. This calculation is generic, meaning it is parameterized by implementations for abstracting random numbers, and unary and binary operators. The parameters for the unary operator case are an abstract unary denotation \[]\] and a proof that it abstracts concrete unary denotation:

\[ \alpha^v((\lambda(V).\{v\mid v \in V\})(\rho^2)) \subseteq \text{eval}^f \rho^2 \]

The calculation for the remaining forms can be found in Cousot's notes [8, lec. 16]. This calculation serves to contrast the concrete calculation we develop in section 4.5, which is more amenable to verification and extraction in Agda.

3. Mechanization: The Easy Parts

We aim to mechanize the style presented in figure 3. Some parts are easy; we start with those.

Figure 4 gives the syntax and semantics in Agda. Variables are modelled as an indexed into an environment of statically known size; otherwise, the syntax of \text{exp} translates directly. The meaning of unary operators is given by a function, \[\text{eval}^f\], while binary operators are defined relationally, \[\text{eval}^f\], to account for the partiality of \[\text{eval}^f\].

The process of calculation is to construct \text{eval}^f through a chain of ordered reasoning: \(c_{\text{eval}^f}(\text{exp}) = \alpha^v\circ\text{eval}^f\circ\gamma^v\). An abstract semantics \text{eval}^f can then be shown to satisfy this specification through an ordered relationship \(c_{\text{eval}^f}(\text{exp}) \subseteq \text{eval}^f\).

To encode \text{eval}^f, we create powersets using \text{characteristic functions}, assuming set-theoretic primitives (defined later), where the judgement \(x \in \text{mk}[\text{eval}^f]\) holds if \(\phi(x)\) is inhabited.

\[ \begin{align*}
  \phi : \text{Set} & \rightarrow \text{Set} \\
  \text{mk}[\phi] & : \forall [A] \rightarrow (A \rightarrow \text{Set}) \rightarrow \phi A \\
  \_ \in & : \forall [A] \rightarrow A \rightarrow \phi A \rightarrow \text{Set}
\end{align*} \]

The \text{eval} function is then defined using an existential type inside of a characteristic function:

\[ \begin{align*}
  \text{eval}^f & : \forall [\gamma] \rightarrow \text{exp} \rightarrow \text{eval}^f \rightarrow \phi \text{eval}^f \rightarrow \gamma \text{eval}^f \\
  \text{eval}^f & = \text{mk}[\text{eval}^f] (\lambda(v \rightarrow \exists p \text{ st } (\rho \vdash p \mapsto v))
\end{align*} \]

4. Constructive Abstract Interpretation

The Galois connections presented in the section 2 are not immediately amenable to encoding in Agda, or constructive logic in general. The heart of the problem is the definition of \(\alpha^v\):

\[ \alpha^v(V) \equiv \begin{cases} 
  \bot & \text{if } \exists v \in V : v < 0 \\
  0 & \text{if } 0 \in V \\
  + & \text{if } \exists v \in V : v > 0
\end{cases} \]

2015/7/13
A literal translation of $\alpha^2$ to constructive logic would require deciding predicates such as $\exists \phi \in V : \phi \not< 0$ in order to return a value of type $\mathbb{N}^2$, however such predicates are in general undecidable.

There are a number of known options for encoding $\alpha^2$, each of which has shortcomings for our goal of extracting computation from the result of a verified calculation.

**Non-solution 1: Admit Excluded Middle** One option to defining $\alpha^2$ is to postulate the law of excluded middle:

\[
\text{excluded--middle} : \forall (P : \text{Set}) \rightarrow P \lor (\neg P)
\]

This axiom imbeds the logic with classical reasoning, is logically consistent, and would allow us to perform case analysis on the existential predicate $\exists x \in V : v < 0$ to complete a definition for $\alpha^2$. This approach has the drawback that definitions no longer carry computational content, and cannot be extracted or computed with in general.

**Non-solution 2: Work in Powerset** Another option is to always work inside the powerset type $\wp$, meaning $\alpha^2$ would have type $\wp(\wp(\mathbb{N}))$. This approach also allows for a successful definition of $\alpha^2$, but again suffers from not being a computation. Functions at type $\wp(\wp(\mathbb{N})) \rightarrow \wp(\wp(\mathbb{N}))$ cannot be executed to produce values at type $\wp(\mathbb{N})$, which is the end goal.

**Non-solution 3: Only use Concretization** The state of the art in mechanizing abstract interpreters is to use “$\gamma$-only” encodings of soundness and completeness properties [14]. However, this approach has a number of drawbacks: it does not support calculation, it gives the engineer no guidance as to whether or not their $\gamma$ is sensible (sound and complete w.r.t. $\alpha$), and it is less compositional than the Galois connection framework.

4.1 Our Solution: A Specification Effect

The problem of encoding Galois connections in constructive logic exists with an apparent dichotomy: if the construction is too classical then one cannot extract computation from the result, and if it is too constructive it prevents the definition of classical structures like Galois connections. We find a solution to this problem through a new Galois connection framework which marks the transition from constructive to classical with a monadic effect. Classical and constructive reasoning can then be combined within the same framework, and classical constructions can be promoted to constructive ones after they are shown to be effect-free.

We find a solution to the problem of encoding calculational abstract interpretation in constructive logic by reformulating the definition of a Galois connection into the powerset Kleisli category. This approach:

1. is rooted in the first principles of Galois connections,
2. allows for the definition of Galois connections which would otherwise require classical reasoning (like excluded middle),
3. supports abstract interpretation by calculus, and
4. allows for the extraction of algorithms from calculations.

The transition to the powerset Kleisli category results in abstraction and concretization maps which have a specification effect, meaning they return a classical powerset value, which we model non-constructively. The laws that accompany the Galois connection will then introduce and eliminate this effect. Combined with monad laws, which also introduce and eliminate monadic effects, we are able to mix constructive and classical reasoning and extract an algorithm from the result of calculation, after all introduced effects have been eliminated.

4.2 Kleisli Galois Connections

Kleisli Galois connections are formed by re-targeting the classical Galois connection framework from the category of posets to the powerset Kleisli category. The morphisms in this category are monotonotonic monadic functions $A \rightarrow \wp(B)$ rather than their classical counterparts $A \rightarrow B$. Powersets $\wp(A)$ are required to be monotonotonic themselves, meaning they are downward closed, i.e. $X \in \wp(A)$ is monotonotonic if and only if $\forall (x, y) . x \in Y \rightarrow Y \subseteq X \rightarrow y \in X$.

The reflexive morphism in the powerset Kleisli category is return, rather than id, where return is defined as the downward
Numeric literals
\[ α^\text{num} \ast (\text{eval}^\text{num}[n] \ast (\gamma^\text{attr}(p^\delta))) \]
\[ = α^\text{num} \ast (\{v \mid \exists \rho \in \gamma^\text{attr}(p^\delta) : \rho \vdash n \mapsto v\}) \]
\[ \subseteq α^\text{num} \ast (\text{return}(n)) \]
\[ = α^\text{num}(n) \]
\[ = \text{return}(\eta(n)) \]
\[ \doteq \text{return}(\text{eval}^\text{num}[n](p^\delta)) \]

Variable references
\[ α^\text{var} \ast (\text{eval}^\text{var}[x] \ast (\gamma^\text{attr}(p^\delta))) \]
\[ = α^\text{var} \ast (\{v \mid \exists \rho \in \gamma^\text{attr}(p^\delta) : \rho \vdash x \mapsto v\}) \]
\[ = α^\text{var} \ast (\{\rho(x) \mid \rho \in \gamma^\text{attr}(p^\delta)\}) \]
\[ = \text{return}(\lambda(\rho).\text{return}(\rho(x))) \ast (\gamma^\text{attr}(p^\delta)) \]
\[ \subseteq \text{return}(p^\delta(x)) \]
\[ \doteq \text{return}(\text{eval}^\text{var}[x](p^\delta)) \]

Unary operators
\[ α^\text{un} \ast (\text{eval}^\text{un}[e] \ast (\gamma^\text{attr}(p^\delta))) \]
\[ = α^\text{un} \ast (\{v \mid \exists \rho \in \gamma^\text{attr}(p^\delta) : \rho \vdash e \mapsto v\}) \]
\[ = α^\text{un} \ast (\{\rho(x) \mid \rho \in \gamma^\text{attr}(p^\delta)\}) \]
\[ = \text{return}(\lambda(\rho).\text{return}(\rho(x))) \ast (\gamma^\text{attr}(p^\delta)) \]
\[ \subseteq \text{return}(p^\delta(\lambda(x).\text{return}(\rho(x)))) \]
\[ \doteq \text{return}(\text{eval}^\text{un}[e](p^\delta)) \]

The monadic bind operator, which we call extension and note \ast, in the tradition of Moggi [18], is defined using a dependent sum, or existential type:

\[ \_ \ast \in \forall(A,B).(A \rightarrow \wp(B)) \rightarrow (\wp(A) \rightarrow \wp(B)) \]
\[ f \ast (X) = \{y \mid \exists x \in X : y = f(x)\} \]

To establish that \(\wp\) forms a monad with return and \ast we prove left-unit, right-unit and associativity laws.

**Lemma 7** (\(\wp\)-monad). \(\wp\) forms a monad with return and \ast, meaning the following properties hold:

- **left-unit**: \(\forall(X).\text{return} \ast (X) = X\)
- **right-unit**: \(\forall(f,x).f \ast (\text{return}(x)) = f(x)\)
- **associativity**: \(\forall(f,g,X).g \ast (f \ast (X)) = (\lambda(x).g \ast (f(x))) \ast (X)\)

Composition in the powerset Kleisli category is notated \(\_ \odot \_\) and defined with \ast:

\[ \_ \odot \_ \in \forall(A,B,C).(B \rightarrow \wp(C)) \rightarrow (A \rightarrow \wp(B)) \rightarrow A \rightarrow \wp(C) \]
\[ (g \odot f)(x) = g \ast (f(x)) \]

A Kleisli Galois connection \(A \xrightarrow{\alpha^m} B\), which we always note with superscripts \(\alpha^m\) and \(\gamma^m\), is analogous to that of classical Galois connection but with monadic morphisms, unit and composition:

\[ \alpha^m \in A \rightarrow \wp(B) \]
\[ \gamma^m \in B \rightarrow \wp(A) \]
\[ \text{extension}^m : \forall(x).\text{return}(x) \subseteq \gamma^m \ast (\alpha^m(x)) \]
\[ \text{reductive}^m : \forall(x^\delta).\alpha^m \ast (\gamma^m(x^\delta)) \subseteq \text{return}(x^\delta) \]

The presence of return as the identity is significant: return marks the transition from pure values to those which have a “specification effect”. extension^m states that \(\gamma^m \circ \alpha^m\) is a pure computation at best, and reductive^m states that \(\alpha^m \circ \gamma^m\) is a pure computation at worst. The consequence of this will be important during calculation: appealing to extension^m and reductive^m will introduce and eliminate the specification effect, respectively.

### 4.3 Lifting Kleisli Galois Connections

The end goal of our calculation is stated as a partial order relationship using a classical Galois connection: \(\alpha^{	ext{ext}}(\text{eval}) \subseteq \wp^{\text{ext}}\). If we wish to work with Kleisli Galois connections, we must build bridges between Kleisli results and classical ones. This bridge is stated as an isomorphism between Kleisli Galois connections and a subset of classical Galois connections that hold computational content, as shown in section 1 figure 1. In addition to the Galois connections themselves, we map proofs of relatedness between Kleisli and classical Galois connections, so long as the classical result is of the form \(\alpha^{\text{ext}}(f^\delta) \subseteq f^\delta\) where \(f\) and \(f^\delta\) are monadic functions.

In order to leverage Kleisli Galois connections for our calculation we must recognize eval as the extension of a monotonic
monadic function $eval^m$. Recall the definition of $eval$:

$$eval[e] \in \wp(\text{env}) \quad eval^m[e] \in env \rightarrow \wp(\text{val})$$

This is the extension of the monadic powerset function: $eval^m$:

$$eval^m[e] \in env \rightarrow \wp(\text{val})$$

where, by definition of $\_^*$:

$$eval^m[e](\rho) := \{v | \rho \vdash e \Rightarrow v\}$$

This observation enables us to construct a Kleisli Galois connection:

$$\text{env} \rightarrow \wp(\text{val}) \xleftarrow{\gamma} env^\sharp \rightarrow \wp(\text{val}^\sharp)$$

and calculation

$$\alpha^{\text{spec}}(eval^m[e]) \subseteq eval^m[e]$$

and lifts the results to classical ones automatically via the soundness of the mapping from Kleisli to classical. Furthermore, we know that any classical Galois connection and classical calculation of $eval^m$ can be encoded as Kleisli via the completeness of the Kleisli to classical mapping. We give precise definitions for soundness and completeness in section 7.

4.4 Constructive Galois Connections

When performing calculation to discover $eval^{m^t}[e]$ from the induced specification $\alpha^{m^t}(eval^m[e])$, we will require that the result be an algorithm, which we can now state precisely as having no monadic effect. The goal will then be to calculate the pure function $eval^{m^t}[e] \in env^\sharp \rightarrow \text{val}^\sharp$ such that

$$\alpha^{m^t}(eval^m[e])(\rho^\sharp) \subseteq return(eval^{m^t}[e])(\rho^\sharp)$$

However, at present, such a calculation will be problematic. Take for instance, the definition we would like to end up with for numeric literal expressions:

$$eval^{m^t}[n](\rho^\sharp) := \alpha^{m^t}(n)$$

This defines the abstract interpretation of a numeric literal as the immediate lifting of that literal to an abstract value. However, this definition is not valid, since $\alpha^{m^t} \in \text{val} \rightarrow \wp(\text{val}^\sharp)$ introduces a specification effect. The problem becomes magnified when we wish to parameterize over $\alpha^{m^t}$, as Cousot does in his original derivation.

One idea is to restrict all $\alpha^{m^t}$ mappings to be pure, and only parameterize over abstractions for $\text{val}$ which have pure mappings. We take morally this approach, although later we show that it is not a restriction at all, and arises naturally through an isomorphism between Kleisli Galois connections and those which have pure abstraction functions, which we call constructive Galois connections. This isomorphism is visualized on the right-hand-side of figure 1, and proofs are given in section 7.

A constructive Galois connection is a variant of Kleisli Galois connection where the abstraction function $\alpha^{m^t}$ is required to have no specification effect, which we call $\eta$ following the convention of [19, p. 237] where it is called an “extraction function”:

$$\eta : A \rightarrow B$$

$$\gamma^c : (\lambda(x).\text{return}(\eta(x))) \cdot \gamma(x^2) \subseteq return(x^2)$$

We can now define the abstract interpretation for numeric literals as:

$$eval^{m^t}[n](\rho^\sharp) := \eta^c(n)$$

which is a pure computation that can be extracted and executed.

4.5 Calculating the Interpreter, Constructively

We now recast Cousot’s calculational derivation of a generic abstract interpreter in the setting of Kleisli Galois connections. In the next section we show how the constructive version is translatable to Agda. As before, we only show numeric literals, variable reference and unary operators; see our full Agda development for constructive calculations of the remaining cases.

Recall the constructive calculation goal, which is to discover a pure function $eval^{m^t}$ such that

$$\alpha^{m^t}(eval^m)[\rho^\sharp] \subseteq return(eval^{m^t}[\rho^\sharp])$$

This goal makes it clear that we are starting with a specification $eval^m : env \rightarrow \wp(\text{val})$, and working towards a pure computation $eval^{m^t} : env^\sharp \rightarrow \text{val}^\sharp$. The process of calculation will eliminate the “specification effect” from the induced specification $\alpha^{m^t}(eval^m)$ using monadic laws and the reductive and expansive properties of Kleisli Galois connections.

The setting assumes Kleisli Galois connections for the abstractions of values $\text{val} \xleftarrow{\gamma^v} \text{val}^\sharp$, environments $\text{env} \xleftarrow{\gamma^\rho} \text{env}^\sharp$ and their induced classical Galois connection for the monadic function space $\text{val} \rightarrow \wp(\text{env}) \xleftarrow{\gamma^\rho} \text{val}^\sharp \rightarrow \wp(\text{env}^\sharp)$. When needed we replace $\alpha^v(x)$ with an equivalent pure extraction function $\text{return}(\eta(x))$ using the isomorphism between Kleisli and constructive Galois connections.

We begin calculating from the specification $\alpha^{m^t}(eval^m)$ by unfolding definitions:

$$\alpha^{m^t}(eval^m[e])(\rho^\sharp) = (\alpha^{m^t} \circ eval^m[e] \circ \gamma^v)(\rho^\sharp)$$

$$\alpha^{m^t} \circ eval^m[e] \circ \gamma^v$$

And proceed by induction on $e$. The calculations for numeric literals, variables and unary operators are shown in figure 4. The parameters for the unary operator case in the constructive setting are an abstract unary denotation $\llbracket \_\rrbracket^\sharp \in \text{val}^\sharp \rightarrow \text{val}^\sharp$ and a proof that it abstracts concrete unary denotation:

$$\alpha^{m^t}(\lambda(x).\text{return}(\llbracket u \rrbracket^\sharp))(\rho^\sharp) \subseteq return(\llbracket u \rrbracket^\sharp(\rho^\sharp))$$

The biggest difference between our constructive derivation and Cousot’s classical derivation is the presence of monadic unit return and extension operator $\_\_$. In the process of calculation, monadic unit and associativity laws are used in combination with extensive and reductive properties to calculate toward a pure value, at which point the result is both a pure computation and an abstraction of $eval[e]$ simultaneously by construction.

5. Galois Connection Metatheory in Agda

We now encode our constructive calculation of the Generic Abstract Interpreter in Agda, both to verify the results mechanically and to extract an executable version of the resulting abstract interpreter. We mechanize the calculation of the interpreter first by developing a theory of posets, its monotonic function space, and a non-constructive powerset type, which we prove is a monad. Then we develop theories of classical, Kleisli and constructive Galois connections, as well as their soundness and completeness relationships. Finally, we embed the constructive calculation in Agda, arriving at an executable algorithm, and lift its correctness property to the classical correctness criteria initially specified by Cousot.

5.1 Posets in Agda

We begin by defining PreOrd, a relation which is reflexive and transitive. PreOrd is a type class, meaning top-level instance definitions will be automatically selected by Agda during type inference.
We then define posets in Agda:

\[
\text{dom} : \text{Poset} \to \text{Set}
\]
\[
\text{dom} \uplus (A \uplus \text{Poset}) = A
\]

\[
data \uplus \text{Poset} : \text{Set} \text{ where}
\]
\[
\text{dom} : \text{Poset} \to \text{Set}
\]
\[
\text{dom} \uplus (A \uplus \text{Poset}) = A
\]

The reason for introducing a new datatype \( \uplus \) is purely technical; it allows us to block reduction of elements of \( \uplus \) until we witness its lifting from a value \( x : \text{dom} A \) into \( \uplus x \text{ Poset} A \).

Next, we induce a partial order on Poset from the antisymmetric closure of the supplied pre order lifted to elements of \( \uplus \):

\[
\text{data} \_\leq \text{dom} : \forall (A : \text{Poset}) \to A \to \text{dom} A \to \text{Set}
\]
\[
\text{data} \_\leq (A \uplus \text{Poset}) : \forall (A : \text{Poset}) \to (A \to A) \to \text{Set}
\]

This definition of \( \_\leq \) is designed to also block reduction until the liftings of \( x \) and \( y \) are likewise witnessed through pattern matching.

We induce equivalence as the antisymmetric closure of \( \subseteq \):

\[
\text{data} \_\equiv \subseteq (A : \text{Poset}) (x y : (A : \text{Poset})) : \text{Set}
\]
\[
\text{data} \_\equiv \subseteq (A : \text{Poset}) (x y : (A : \text{Poset})) : \text{Set}
\]

We prove reflexivity, transitivity and antisymmetry for \( \_\subseteq \), as well as reflexivity, transitivity and symmetry for \( \_\equiv \).

5.2 Monotonic Functions in Agda

To construct a poset for monotonic functions we carry proofs of monotonicity around with each function.

\[
data \text{mon} (A B : \text{Poset}) : \text{Set} \text{ where}
\]
\[
\text{mon} (A B : \text{Poset}) : \text{Set} \text{ where}
\]

The PreOrd for mon is the pointwise ordering of \( \subseteq \):

\[
data \_\leq \text{mon} \subseteq (A B : \text{Poset}) : \forall (A B : \text{Poset}) \to \text{mon} A B \to \text{mon} B A \to \text{Set}
\]
\[
data \_\leq \text{mon} \subseteq (A B : \text{Poset}) : \forall (A B : \text{Poset}) \to \text{mon} A B \to \text{mon} B A \to \text{Set}
\]

5.3 Monotonic Powerset in Agda

We define powersets as monotonic characteristic functions into Agda’s Set type.

\[
data \text{pow} (A : \text{Poset}) : \text{Set} \text{ where}
\]
\[
data \text{pow} (A : \text{Poset}) : \text{Set} \text{ where}
\]

Whereas \( \text{mk}[\_\Rightarrow] \) constructs a monotonic function from \( f \), \( \text{mk}[\_\Rightarrow] \) constructs a set from a monotonic characteristic function \( \_\Rightarrow \), not necessarily monotonic. Antitonicity of the argument to \( \_\Rightarrow \) and \( \_\Rightarrow \) is required to ensure sets are downward closed.

The preorder for pow is implication:

\[
data \_\leq \text{pow} \subseteq (A : \text{Poset}) : \forall (A : \text{Poset}) \to A \to \text{pow} A \to \text{Set}
\]
\[
data \_\leq \text{pow} \subseteq (A : \text{Poset}) : \forall (A : \text{Poset}) \to A \to \text{pow} A \to \text{Set}
\]

The set-containment judgement is \( \subseteq \):

\[
\_\subseteq : \forall (A : \text{Poset}) \to A \to \text{pow} A \to \text{Set}
\]
\[
\_\subseteq : \forall (A : \text{Poset}) \to A \to \text{pow} A \to \text{Set}
\]

And like functions we provide a cheat for creating monotonic sets without the burden of monotonicity proof.

\[
data \_\Rightarrow \text{pow} \subseteq (A : \text{Poset}) : \forall (A : \text{Poset}) \to \text{pow} A \to \text{Set}
\]
\[
data \_\Rightarrow \text{pow} \subseteq (A : \text{Poset}) : \forall (A : \text{Poset}) \to \text{pow} A \to \text{Set}
\]
We show Agda types for classical, Kleisli and constructive Galois connections.

Finally, we prove (and omit) monads laws analogous to those in Agda module:

```
module S-env (valφ : Set) (⇒val⇒ : val valφ) where

Abstract environments take the form of another list-like inductive datatype:

```
data envφ : size → Set where
  [] : envφ Zero
  _∷_ : V {Gamma} → envφ Gamma → envφ Suc {Gamma}
```

The ordering for envφ is the pointwise ordering:

```
data ≤φ : V {Gamma} → envφ Gamma → envφ Gamma → Set where
  [] ≤φ []
  _∷_ ≤φ _∷_ : V {Gamma} → [ρ₁ ⊑ ρ₂ : envφ Gamma] [v₁ v₂] → v₁ ≤φ v₂ → ρ₁ ≤φ (v₁ ⊓ ρ₁) ≤φ (v₂ ⊓ ρ₂)
```

The environment abstraction function ηφ is the pointwise application of ηφ to val⇒:

```
ηφ : V {Gamma} → envφ Gamma → envφ Gamma
ηφ [] = []
ηφ (n :: ρ) = ηφ val⇒ • (n :: ρ)
```

The concrete function γφ is the pointwise concretization of γφ:

```
data ∈γφ : V {Gamma} → envφ Gamma → envφ Gamma → Set where
  [] ∈γφ []
  _∷_ ∈γφ _∷_ : V {Gamma} → [ρ : envφ Gamma] [φ : envφ Gamma] [n φ] → (n ∈γφ val⇒ • φ) ∈γφ (n :: ρ) ∈γφ (n :: ρ)
```

The ∈γφ and ∈γφ functions are sound and complete by pointwise applications of soundness and completeness from val⇒:

```
soundφ : V {Gamma} → [ρ : envφ Gamma] → ρ ∈γφ ηφ ρ
soundφ [] = []
soundφ (x :: ρ) = soundφ val⇒ • sound⇒ ρ
```

```
completeφ : V {Gamma} → [ρ : envφ Gamma] → ρ ∈γφ ρ ≤φ pφ
completeφ [] = []
```

```
completeφ (n ρ) = completeφ val⇒ • complete⇒ (n ρ)
```

5.4 Powerset Monad in Agda

The powerset monad has unit return, where return x is the set of all elements smaller than x, as defined by a characteristic function:

```
return : V A {Poset} → A ⇒ φ A
return = mk⇒ (λ x → mk[k] (λ y → y ≤ x))
```

We lift the return operation to functions, which we call pure.

```
pure : V (A B : Poset) → A ⇒ φ B → A ⇒ φ B
pure = mk⇒ (λ f → mk[k] (λ x → return • (f x))
```

Monadic extension is _∗_.

```
_*_ : V (A B : Poset) → A ⇒ φ B → A ⇒ φ B
_*_ = mk⇒ (λ X → mk[k] (λ y → ∃ x st (x ∈ X) × x ∈ f • x))
```

We use _*_ to define Kleisli composition, _⊙_:

```
_⊙_ : V (A B C : Poset) → B ⇒ φ C → A ⇒ φ B → A ⇒ φ C
g _⊙_ f = mk⇒ (λ x → g • (f x))
```

Finally, we (prove and omit) monads laws analogous to those in lemma 7.

6. Calculational Abstract Interpretation in Agda

We show Agda types for classical, Kleisli and constructive Galois connections in figure 6. Using these definitions we calculate an abstract interpreter in Agda following the constructive approach described in section 4 in the following steps:

1. Define a constructive Galois connection between env and envφ.
From these definitions, we construct \(\forall \Gamma \rightarrow ((\text{env } \Gamma) \times (\text{env } \Gamma)) \vdash \gamma \times \gamma \times (\text{env } \Gamma)\) using a helper function \(\text{mk}[\text{expr}]\) for lifting primitive definitions (non-\(\text{POSet}\)) to Galois connections.

\[
\begin{align*}
\text{zenv} : \forall \Gamma \rightarrow ((\text{env } \Gamma) \times (\text{env } \Gamma)) \vdash \gamma \times \gamma \times (\text{env } \Gamma)
\end{align*}
\]

\[
\begin{align*}
\text{zenv} = \text{mk}[\text{expr}] \eta'(\lambda \rho^2 \rightarrow \rho^2) \text{ sound and complete*}
\end{align*}
\]

6.2 Inducing a Best Specification in Agda

The monadic semantics is encoded with the evaluation relation:

\[
\begin{align*}
\text{eval}^m[\_] : \forall \Gamma \rightarrow \Gamma \rightarrow \gamma \rightarrow \gamma \rightarrow (\text{val } \Gamma) \\
\text{eval}^m[\_] = \text{mk}[\text{expr}] \eta'(\lambda \rho^2 \rightarrow \rho^2)
\end{align*}
\]

To induce a best abstraction we first encode the pointwise lifting of two Kleisli Galois connections \(\forall \Gamma \rightarrow \text{zval } \Gamma\) to classical pointwise Galois connections over the monadic function space as

\[
\begin{align*}
\text{zval}^m \rightarrow \forall \Gamma \rightarrow \text{zval } \Gamma \rightarrow \gamma \rightarrow \gamma \rightarrow \text{val } \Gamma
\end{align*}
\]

We can now state the specification for \(\text{eval}^m[\_]\) as a pure function which approximating \(\text{eval}[\_] : \forall \Gamma \rightarrow \text{val } \Gamma \rightarrow \gamma \rightarrow \gamma \rightarrow \text{val } \Gamma\).

6.3 Calculating the Interpreter, in Agda

Before calculating we must lift the various semantic functions to the monoton function space, like \(\_ : \_\) and \(\_ : \_\):

\[
\begin{align*}
\text{lookup}[\_] : \forall \Gamma \rightarrow \gamma \rightarrow \gamma \rightarrow \text{val } \Gamma \\
\text{lookup}[\_] = \text{mk}[\text{expr}] \eta'(\lambda \rho^2 \rightarrow \rho^2)
\end{align*}
\]

\[
\begin{align*}
\text{postulate}
\end{align*}
\]

We now set up to calculate \(\text{eval}^m[\_]\) from its specification \(\text{eval}[\_] : \forall \text{val } \Gamma \rightarrow \gamma \rightarrow \gamma \rightarrow \text{val } \Gamma\). Because we want to “discover” \(\text{eval}^m[\_]\), rather than verify it a-posteriori, we state its existence and then calculate its implementation:

\[
\begin{align*}
\text{eval}^m[\_] : \forall \Gamma \rightarrow \gamma \rightarrow \gamma \rightarrow \text{val } \Gamma
\end{align*}
\]

We begin by stating the type of our calculation:

\[
\begin{align*}
\text{eval}[\_] : \forall \Gamma \rightarrow \gamma \rightarrow \gamma \rightarrow \text{val } \Gamma
\end{align*}
\]

and proceed by induction, the first case being numeric expressions. Each case will make use of our “proof-mode” library, which we have developed in pure Agda to support calculation-style notation.

**Numeric literals** We begin by stating the goal. We do this using our proof mode library with notation \([\_]:\):

\[
\begin{align*}
\text{eval}[\_] = \text{mk}[\text{expr}] \quad \text{eval}[\text{Num } n] \quad \text{eval}[\text{Num } n] \quad \text{rho}^2
\end{align*}
\]

Next we unfold the definition of \(\_ \rightarrow \_\), also by Agda computation:

\[
\begin{align*}
\text{eval}[\text{Num } n] \quad \text{rho}^2
\end{align*}
\]

The next step is to focus to the right of the application and replace \(\text{eval}[\text{Num } n] \quad \text{rho}^2\) with its denotation \(\text{return } \bullet \rightarrow \text{return } \bullet \rightarrow \text{return } \bullet\), which we prove by an auxiliary lemma \(\beta - \text{eval}^m[\_]:\)

\[
\begin{align*}
\text{return } \bullet \rightarrow \gamma \rightarrow \gamma \rightarrow \text{val } \Gamma
\end{align*}
\]

It is at this point where we have reached a pure computation, absent of any specification effect. We declare this expression then to be the definition of \(\text{eval}^m[\text{Num } n]\) and conclude:

\[
\begin{align*}
\text{return } \bullet \rightarrow \gamma \rightarrow \gamma \rightarrow \text{val } \Gamma
\end{align*}
\]

**Variables** The calculation for variables is more interesting, as it doesn’t ignore the environment \(\gamma \rightarrow \gamma \rightarrow \text{val } \Gamma\). We begin again by stating the goal:

\[
\begin{align*}
\text{eval}[\text{Var } x] \quad \text{rho}^2
\end{align*}
\]

As before, the first thing we do is unfold the definition of \(\_\):

\[
\begin{align*}
\text{eval}[\text{Var } x] \quad \text{rho}^2
\end{align*}
\]

Next we focus to the right of the left-most, and left of the rightmost \(\bullet \rightarrow \_\) operator:

\[
\begin{align*}
\text{focus-in \_} \rightarrow \gamma \rightarrow \gamma \rightarrow \text{val } \Gamma
\end{align*}
\]

Here we recognize that the specification for the semantics of \text{Var } x is equivalent to the computation of looking up a variable in the environment, using an auxiliary proof \(\beta - \text{lookup}[\_]:\)

\[
\begin{align*}
\text{focus-in \_} \rightarrow \gamma \rightarrow \gamma \rightarrow \text{val } \Gamma
\end{align*}
\]

Next we exploit the relationship between concrete environment lookup and abstract environment lookup: \(\text{eval}[\text{Var } x] \rightarrow \text{eval}[\text{Var } x]\)
\[ \text{val } z \mapsto \text{(pure } \cdot \text{lookup } [x]) \in \text{pure } \cdot \text{lookup}^r [x] \text{. To arrive at } \alpha [\text{val } z] \mapsto \text{pure } \cdot \text{lookup}^r [x], \text{ we first reason by extensiveness of } \text{val } z:\]

\[\begin{align*}
\&[[ (\text{pure } \cdot \text{lookup } [x] \mapsto (\gamma [\text{env } z] \cdot \rho^r))]
\&\ldots\text{extensive}^\rho[[z val z]] [\gamma [\text{env } z] \cdot \rho^r]
\&[[ \gamma [\text{val } z] \mapsto (\text{(pure } \cdot \eta [\text{val } z]) \mapsto \text{pure } \cdot \text{lookup}^r [x] \mapsto (\gamma [\text{env } z] \cdot \rho^r))]]
\end{align*}\]

We identify the argument to the application as \(\alpha [\text{val } z] \mapsto \gamma [\text{val } z] \mapsto (\text{pure } \cdot \text{lookup } [x])\) and weaken by its abstraction:

\[\begin{align*}
\&\text{focus–right } [] \gamma [\text{val } z] \mapsto (\gamma [\text{val } z] \mapsto (\text{pure } \cdot \text{lookup } [x] \mapsto (\gamma [\text{env } z] \cdot \rho^r)))
\&\ldots\text{reductive}^\rho[[z val z]] [\gamma [\text{val } z] \mapsto (\gamma [\text{val } z] \mapsto (\text{pure } \cdot \text{lookup}^r [x] \mapsto (\gamma [\text{env } z] \cdot \rho^r)))]
\end{align*}\]

Finally we apply the reductive property of \(\text{val } z\):

\[\begin{align*}
\&\text{reduce}^\rho [[z val z] \mapsto (\gamma [\text{val } z] \mapsto (\text{pure } \cdot \text{lookup}^r [x] \mapsto (\gamma [\text{env } z] \cdot \rho^r)))]
\end{align*}\]

and declare the result as defining \(\text{eval}^r \text{Var}_{x} \text{ and conclude:}\)

\[\begin{align*}
\&[[ \text{return } (\text{eval}^r \text{Var}_{x} ) \mapsto (\gamma [\text{val } z] \mapsto (\text{pure } \cdot \text{lookup}^r [x] \mapsto (\gamma [\text{env } z] \cdot \rho^r)))]
\end{align*}\]

**Unary operators** The calculation of unary operators is interesting because it leverages an inductive hypothesis for the calculation.

\[\text{eval}^r (\text{Unary } o [e] \mapsto \rho^r) \text{ with } \text{eval}^r e \mapsto \rho^r \text{. }\]

\[\ldots \text{IH } [\text{proof–model } \alpha [\text{env } z] \mapsto (\text{eval}^r \text{Unary } o [e] \mapsto (\gamma [\text{env } z] \cdot \rho^r))]
\]

In Agda, the notation is a variation of let-binding which also performs dependent pattern matching refinements (although this example doesn’t use dependent pattern matching). We proceed as before by expanding the definition of \(\alpha [\text{].}\)

\[\begin{align*}
\&[[ \text{pure } \cdot \eta [\text{val } z] \mapsto (\text{eval}^r \text{Unary } o [e] \mapsto (\gamma [\text{env } z] \cdot \rho^r))]\]
\&[[ \text{pure } \cdot \eta [\text{val } z] \mapsto (\gamma [\text{env } z] \cdot \rho^r)]]
\end{align*}\]

As before we focus between then \(\ldots\).

\[\begin{align*}
\&\text{focus–right } [] (\text{pure } \cdot \eta [\text{val } z]) \mapsto (\gamma [\text{val } z] \mapsto (\text{eval}^r \text{Unary } o [e] \mapsto (\gamma [\text{env } z] \cdot \rho^r))),
\&\text{focus–left } [] (\gamma [\text{val } z] \mapsto (\text{eval}^r \text{Unary } o [e] \mapsto (\gamma [\text{env } z] \cdot \rho^r)))\]
\end{align*}\]

We then replace the \(\text{eval}^r \text{Unary } o [e] \mapsto (\gamma [\text{env } z] \cdot \rho^r))\) specification with an equivalent computation: pure \(\mapsto \text{pure } \cdot \text{lookup}^r \).

\[\begin{align*}
\&\text{focus–in } [] \begin{cases}
\&[[ \text{pure } \cdot \text{lookup}^r [e] \mapsto (\gamma [\text{env } z] \cdot \rho^r))]
\&[[ \text{pure } \cdot \text{lookup}^r [e] \mapsto (\gamma [\text{env } z] \cdot \rho^r))]
\&[[ \text{pure } \cdot \text{lookup}^r [e] \mapsto (\gamma [\text{env } z] \cdot \rho^r))]
\&[[ \text{pure } \cdot \text{lookup}^r [e] \mapsto (\gamma [\text{env } z] \cdot \rho^r))]
\end{cases}
\end{align*}\]

We then reassociate.

\[\begin{align*}
\&[[ (\text{pure } \cdot \text{lookup}^r [e] \mapsto (\gamma [\text{env } z] \cdot \rho^r))]
\&\ldots \text{extensive}^\rho[[z val z]] [\gamma [\text{env } z] \cdot \rho^r]
\&[[ \gamma [\text{val } z] \mapsto (\text{(pure } \cdot \eta [\text{val } z]) \mapsto \text{pure } \cdot \text{lookup}^r [x] \mapsto (\gamma [\text{env } z] \cdot \rho^r))]]
\end{align*}\]

We focus to the argument of the application and apply extensiveness of \(\text{val } z\):

\[\begin{align*}
\&\text{focus–right } [] (\text{pure } \cdot \text{lookup}^r [e] \mapsto (\gamma [\text{val } z] \mapsto (\text{pure } \cdot \text{lookup}^r [x] \mapsto (\gamma [\text{env } z] \cdot \rho^r))),
\&\text{reduce}^\rho [[z val z] \mapsto (\gamma [\text{val } z] \mapsto (\text{pure } \cdot \text{lookup}^r [x] \mapsto (\gamma [\text{env } z] \cdot \rho^r)))]
\end{align*}\]

We recognize the argument to be \(\alpha [\text{val } z] \mapsto \gamma [\text{val } z] \mapsto (\text{pure } \cdot \text{eval}^r [e] \mapsto \rho^r),\) which we can weaken to \(\text{pure } \cdot \text{eval}^r [e] \mapsto \rho^r\) from the inductive hypothesis:

\[\begin{align*}
\&\text{reduce}^\rho [[z val z] \mapsto (\gamma [\text{val } z] \mapsto (\text{pure } \cdot \text{eval}^r [e] \mapsto \rho^r))]
\end{align*}\]

We apply the monad right unit to eliminate the extension:

\[\begin{align*}
\&\text{right–unit}^\rho [[\gamma [\text{val } z] \mapsto (\text{eval}^r [e] \mapsto \rho^r))]
\&[[ \text{pure } \cdot \text{eval}^r [e] \mapsto (\gamma [\text{val } z] \mapsto (\text{eval}^r [e] \mapsto \rho^r))]
\end{align*}\]

Next we recognize this as an abstraction of \(\text{pure } \cdot \text{eval}^r [e],\) for which we have parameterized the calculation:

\[\begin{align*}
\&[[ \text{pure } \cdot \text{eval}^r [e] \mapsto (\gamma [\text{val } z] \mapsto (\gamma [\text{val } z] \mapsto (\text{eval}^r [e] \mapsto \rho^r))]
\end{align*}\]

We declare the result to be the definition of \(\text{eval}^r\) and conclude:

\[\begin{align*}
\&[[ \text{return } (\text{eval}^r \text{Unary } o [e] \mapsto (\gamma [\text{val } z] \mapsto (\text{eval}^r [e] \mapsto \rho^r)))]
\end{align*}\]

We can then define \(\text{eval}^r\) as the result of calculating:

\[\begin{align*}
\text{eval}^r \text{Num } n = m \mapsto [[\lambda (\rho^r \mapsto \eta [\text{val } z] \mapsto \eta [\text{val } z]))
\text{eval}^r \text{Var } x = m \mapsto [[\lambda (\rho^r \mapsto \eta [\text{val } z] \mapsto \eta [\text{val } z))]
\text{eval}^r \text{Unary } [u] e = m \mapsto [[\lambda (\rho^r \mapsto \eta [\text{val } z] \mapsto \eta [\text{val } z))]
\end{align*}\]

\[4.6 \text{ End to End: Connection to the Collecting Semantics}

Recall that the original collecting semantics we wish to abstract, \text{eval}, is the extension of the monadic semantics, \text{eval}^m. To establish the final proof of abstraction, we promote the partial order of the previous section between monadic functions:
to a partial ordering between extended functions:

\[ \alpha[\text{eval}] : \alpha[\text{env}\xi] \Rightarrow (\text{eval}\xi) \cdot \text{eval}[\xi] + (\text{pure} \cdot \text{eval}[\xi]) + \]

where \( \_ \cdot \text{eval}\xi \) is the promotion operator from Kleisli to classical Galois connections, and \( \Rightarrow \) \( \text{eval}\xi \) is the standard classical Galois connection pointwise lifting operator.

We define \( \_ \cdot \text{eval}\xi \) following the proof of inclusion from Kleisli Galois connections into classical Galois connections:

\[ \_ \cdot \text{eval}\xi : \forall (A_1, A_2 : \text{POSets}) \to A_1 \times A_2 \to (\_\forall (A_1, A_2)) \]

and we prove soundness and completeness following the definitions given in section 7:

\[ \text{sound/complete} : \forall (A_1, A_2, B_1, B_2 : \text{POSets}) \]

\[ (\text{eval}\xi A \Rightarrow (\text{eval}\xi B)) \Rightarrow (\text{eval}\xi (\text{env}\xi A) \Rightarrow (\text{eval}\xi (\text{env}\xi B))) \]

\[ \Rightarrow (\_\forall (\_\forall (\text{eval}\xi A) \Rightarrow (\text{eval}\xi B))) \Rightarrow \text{sound/complete} \]

\[ \Rightarrow \_\forall (\text{eval}\xi A) \Rightarrow (\text{eval}\xi B) \]

Next section describes the soundness and completeness result which we rely on in this section in more detail, and develops the foundations of Kleisli Galois connections.

7. Foundations of Kleisli Galois Connections

Kleisli Galois connections are formed by re-targeting the classical Galois connection framework from the category of posets to the powerset Kleisli category, where morphisms are monomonic monadic functions, as described in section 4.2.

Unfolding the definition of \( \text{extensive}^{\alpha} \) and \( \text{reductive}^{\alpha} \) from section 4.2 we reveal equivalent, more intuitive properties, which we call \( \text{soundness}^{\alpha} \) and \( \text{completeness}^{\alpha} \):

\[ \text{soundness}^{\alpha} : \forall (x, y : \alpha) \wedge y \in \gamma\alpha(x) \Rightarrow x \in \gamma\alpha(y) \]

\[ \text{completeness}^{\alpha} : \forall (x, y : \alpha) \wedge x \in \gamma\alpha(y) \Rightarrow x \subseteq y \]

These definitions provide a relational setup for the soundness and completeness of \( \alpha^{\alpha} \) and \( \gamma^{\alpha} \). In fact, the model for the monadic space \( A \rightarrow \phi(B) \) is precisely \( A \rightarrow \phi \rightarrow \text{Set} \), i.e. the monomonic relations over \( A \) and \( B \). We have therefore recovered a relational setting for soundness and completeness of abstractions between sets, purely by following the natural consequences of instantiating the Galois connection framework to the powerset Kleisli category.

7.1 Lifting Kleisli Galois Connections

The final step of our calculational relies on bridging the gap between Kleisli and classical Galois connections. This bridge enables us to construct a Kleisli Galois connection

\[ \text{env} \Rightarrow (\text{eval}\xi) \Rightarrow (\text{eval}\xi) \cdot \text{eval}[\xi] + (\text{pure} \cdot \text{eval}[\xi]) + \]

and calculation \( \alpha^{\alpha}\gamma^{\alpha}\text{eval}\xi \subseteq \text{eval}\xi \cdot \text{eval}[\xi] + (\text{pure} \cdot \text{eval}[\xi]) + \) and lift both systematically to classical results, and to do so without any loss of generality. We formalize these notions in the following theorems:

**Theorem 1 (GC-Soundness).** For every Kleisli Galois connection

\[ A \xrightarrow{\alpha^{\alpha}} B \]

there exists a classical Galois connection

\[ \phi(A) \xrightarrow{\gamma^{\alpha}} \phi(B) \]

where \( \alpha^{\gamma} \) and \( \gamma^{\alpha} \) are the form \( \alpha^{\gamma} = \gamma^{\alpha} \).

**Theorem 2 (GC-Completeness).** For every classical Galois connection

\[ \phi(A) \xrightarrow{\gamma^{\alpha}} \phi(B) \]

where \( \alpha \) and \( \gamma \) are of the form \( \alpha^{\gamma} = \gamma^{\alpha} \) and \( \gamma^{\alpha} = \gamma^{\alpha} \), there exists a Kleisli Galois connection

\[ A \xrightarrow{\alpha^{\alpha}} B \]

Next we show how to lift Kleisli Galois connections pointwise to a classical Galois connection over extensions:

**Lemma 8 (Pointwise-lifting-extensions).** Given Kleisli Galois connections

\[ A \xrightarrow{\gamma^{\alpha}} B \]

there exists a classical Galois connection

\[ \phi(A) \xrightarrow{\gamma^{\alpha}} \phi(B) \]

where \( \alpha^{\gamma} \) and \( \gamma^{\alpha} \) are the form \( \alpha^{\gamma} = \gamma^{\alpha} \).

And finally we establish an isomorphism of partial ordering between monadic functions and their extensions:

**Theorem 3 (Soundness).** Given Kleisli Galois connections

\[ A \xrightarrow{\gamma^{\alpha}} B \]

and functions \( f \in A \rightarrow \phi(B) \) and \( f^{\alpha} \in A \rightarrow \phi(B) \), partial orders under the Kleisli pointwise lifting imply partial orders under extension:

\[ \alpha^{\gamma^{\alpha}}(f^{\alpha}) \subseteq f^{\alpha} \Rightarrow \alpha^{\gamma^{\alpha}}(f^{\alpha}) \subseteq f^{\alpha} \cdot \]

**Theorem 4 (Completeness).** Given Kleisli Galois connections

\[ A \xrightarrow{\gamma^{\alpha}} B \]

and functions \( f \in A \rightarrow \phi(B) \) and \( f^{\alpha} \in A \rightarrow \phi(B) \), partial orders under the Kleisli pointwise lifting for extensions imply partial orders without extension:

\[ \alpha^{\gamma^{\alpha}}(f^{\alpha}) \subseteq f^{\alpha} \Rightarrow \alpha^{\gamma^{\alpha}}(f^{\alpha}) \subseteq f^{\alpha} \cdot \]
7.2 Constructive Galois Connections

Analogously to Kleisli Galois connection, we state extensiveness and reductiveness as equivalent soundness and completeness properties:

soundness : \( \forall (x), x \in \gamma(\eta(x)) \)

completeness : \( \forall (\gamma^e, x), x \in \gamma^e(\gamma^i) \Rightarrow \eta(x) \subseteq x^e \)

These statements have even stronger intuitive meaning that that of Kleisli Galois connections. soundness states that \( x \) must be in the concretization of its abstraction, and completeness states that the best abstraction for \( x \), i.e. \( \eta(x) \), must be better any other abstraction for \( x \), i.e. \( x^e \).

Constructive Galois connections are initially motivated by the need for pure abstraction functions during the process of calculation, and simultaneously from the observation that abstraction functions are often pure function in practice. What is surprising is that constructive Galois connections are not a special case of Kleisli Galois connections: all Kleisli Galois connections are constructive.

**Theorem 5.** The set of Kleisli Galois connections is isomorphic to the set of constructive Galois connections.

**Proof.** The easy direction is constructing a Kleisli Galois connection from a constructive Galois connection. Given a constructive Galois connection \( A \xleftarrow{\eta} B \), we construct the following Kleisli Galois connection:

\[
\begin{align*}
\alpha^m : A &\rightarrow \wp(B) \\
\gamma^m : B &\rightarrow \wp(A) \\
\alpha^m &= \text{pure}(\eta) \\
\gamma^m &= \gamma^c
\end{align*}
\]

Proofs for extensiveness and reductiveness follow definitionally.

The next step is to construct a Constructive Galois connection from a Kleisli Galois connection \( A \xleftarrow{\eta} B \). This at first seems paradoxical, since it requires constructing an abstraction function \( \eta : A \rightarrow B \) from the given abstraction specification \( \alpha^m : A \rightarrow \wp(B) \). However, we are able exploit the property of soundness, which is equivalent to extensivity, from the definition of Kleisli Galois connections to define \( \gamma \).

Recall the soundness judgement for Kleisli Galois connections, which arise directly from the definition of *return* and _.*

\[
\text{soundness}^m : \forall (x), \exists(y), y \in \alpha(x) \wedge x \in \gamma(y)
\]

Given a proof of soundness, we use the axiom of choice to extract the existentially quantified \( y \) given an \( x \). In fact, the axiom of choice is not an axiom in constructive logic, rather it is a *theorem* of choice, which can be written in Agda.

```
choice : \forall \{A \times B \} \; \{P : A \rightarrow B \rightarrow \text{Set} \} \rightarrow (\forall x \rightarrow \exists y \; \text{st} \; P x y) \rightarrow A \rightarrow B
choice f x with f x
... \Rightarrow \exists y, P[x, y] = y
```

Using the axiom of choice we easily define \( \eta \) and \( \gamma^c \).

\[
\begin{align*}
\eta(x) &= y \text{ where } (\exists y : y \in \alpha^m(x) \wedge x \in \gamma^m(y)) \\
\gamma^c(x) &= \gamma^m(x)
\end{align*}
\]

In order for \( \eta \) and \( \gamma^c \) to be a valid Galois connection we must still prove extensiveness and reductiveness. To do so we instead prove soundness and completeness, which are equivalent to extensivity and reductivity. These proofs follow from the soundness evidence attached to \( \eta(x) \) and its use of the axiom of choice.

**Lemma 9 (soundness).** \( \forall (x), x \in \gamma^c(\eta(x)) \).

**Lemma 10 (completeness).** \( \forall (\gamma^e, x), x \in \gamma^e(\gamma^i) \rightarrow \eta(x) \subseteq x^e \).

Finally, to establish the isomorphism, we show that transforming a Kleisli Galois connection into a constructive one and back results in the same Galois connection. To show this we apply the following lemma, a restatement of its classical analogue [19, p.239] in the Kleisli setting:

**Lemma 11 (Kleisli-Uniqueness).** Given two Kleisli Galois connections \( A \xleftarrow{\eta_1} B \) and \( A \xleftarrow{\eta_2} B \), \( \alpha^m = \alpha^m_2 \) and only if \( \gamma^m_1 = \gamma^m_2 \).

To use this lemma, we recognize that the concretization functions \( \gamma^m_1 \) and \( \gamma^m_2 \) are definitionally the same for both mappings between Kleisli and constructive Galois connections. It then follows that \( \alpha^m_1 \) and \( \text{pure}(\eta_1) \) must be equal.

The consequence of the isomorphism between Kleisli and constructive Galois connections is that we may work directly with constructive Galois connections without any loss of generality. Furthermore, we can assume a pure "extraction function" \( \eta \) for every Kleisli abstraction function \( \alpha^m \) where \( \beta^m = \text{pure}(\eta) \).

Finally, our proof of isomorphism gives a foundational explanation for why some Galois connections happen to have fully computational functions as their abstraction functions. These pure abstraction functions are no accident; they are induced by the Kleisli Galois connection setup embedded in constructive logic, where the axiom of choice is definable as a theorem with computational content.

8. Related Work

This work connects two long strands of research: abstract interpretation and dependently typed programming. The former is founded on the pioneering work of Cousot and Cousot [9, 10]; the latter on that of Martin-Löf [15], embodied in Norell’s Agda [20]. A key technical insight of ours is to use a monadic structure for Galois connections and proofs by calculus, following the example of Moggi [18] for the \( \lambda \)-calculus.

**Calculational abstract interpretation** Cousot describes the calculational approach to abstract interpretation by example in his lecture notes [8], the foundations for which can be found in [7], and recently introduced a unifying calculus for Galois connections [12]. Other notable uses of calculational abstract interpretation include the calculational derivation of higher order control flow analysis [16] and the calculus of polynomial time graph algorithms [23].

Our work mechanizes Cousot’s calculations, and provides a suitable foundation for mechanizing other instances of calculational abstract interpretation.

**Calculational program design** Related to the calculation of abstract interpreters is the calculation of programs, long advocated by Bird and others as calculational program design [2, 3].

Calculational program design has been successfully mechanized in proof assistants [26]. This practice does not encounter the non-constructive metatheory issues which show up in mechanizing calculational abstract interpreters. In mechanized calculational program design, specifications are fully constructive, whose inhabitants can be readily executed as programs. In abstract interpretations the specifications are inherently non-constructive, which leads to the need for new theoretical machinery.

**Verified static analyses** Verified abstract interpretation has seen many promising results [1, 4, 6, 22], scaling up recently to large-scale real-world static analyzers [14]. Mechanized abstract interpretation has yet to benefit from being built on a solid, compositional Galois connection framework. Until now approach have used either “\( \alpha \)-only” or “\( \gamma \)-only” encodings of soundness and (sometimes) completeness. Our techniques for isolating specification effects should readily apply to these existing approaches.
Monadic abstract interpretation  The use of monads in abstract interpretation has recently been used to good effect [13, 24]. However that work uses monads to structure the language semantics, whereas our approach has been to use monadic structure in the Galois connections and proofs by calculus.

Calculator  The Calculator [25] 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; additionally it fully automates the calculational derivations themselves. 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. We foresee a marriage of the two approaches, where simple algebraic calculations are derived automatically, yet more complicated connections are still expressible and provable within the same mechanized framework.

Future directions  Now that we have established a foundation for constructive Galois connection calculation, we see value in verifying larger derivations (e.g. [17, 23]). 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.

There have also been recent developments on compositional abstract interpretation frameworks [13] where abstract interpreter implementations and their proofs of soundness via Galois connection are systematically derived side-by-side. Their framework relies on correctness properties transported by Galois transformers, which we believe would greatly benefit from mechanization, because they hold both computational and specification content.

9. Conclusions

Over fifteen years ago, when concluding “The calculational design of a generic abstract interpreter” [7, p. 85], Cousot wrote:

> 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.

This paper realizes that vision, giving the first mechanically verified proof of correctness for Cousot’s abstract interpreter. Our proof “by calculus” closely follows the original paper-and-pencil proof. The primary discrepancy being the use of monadic reasoning to isolate specification effects. By maintaining this monadic discipline, we are able to verify calculations by Galois connections and extract computation content from pure results. The resulting static analyzer is correct by verified construction and therefore does not suffer from bugs present in the original. ²

Acknowledgments

We gratefully acknowledge the Colony Club in Washington, DC for providing a fruitful environment in which to do this research.

²http://www.di.ens.fr/~cousot/aismart/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] 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.

[5] A. Bove, P. Dybjer, and U. Norell. A brief overview of agda – A functional language with dependent types. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, Theorem Proving in Higher Order Logics, Lecture Notes in Computer Science, chapter 6. Springer Berlin Heidelberg, 2009.

[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. 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.

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

[9] 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.

[10] 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.

[11] P. Cousot and R. Cousot. Basic Concepts of Abstract Interpretation, pages 359–366. Kluwer Academic Publishers, 2004.

[12] 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.

[13] D. Darais, M. Might, and D. Van Horn. Galois transformers and modular Coq interpreters. CoRR, abs/1411.3962, 2014. URL http://arxiv.org/abs/1411.3962

[14] 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.

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

[16] 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.

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

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

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

[20] 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, September 2007.

[21] U. Norell. Dependently typed programming in agda. In P. Koopman, R. Plasmeijer, and D. Swierstra, editors, Advanced Functional Pro-
[22] D. Pichardie. Interprétation abstraite en logique intuitionniste: extraction d’analyseurs Java certifiés. PhD thesis, Université Rennes, 2005.

[23] 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.

[24] 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.

[25] 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.

[26] 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.