Galois Connections for Recursive Types *

Ahmad Salim Al-Sibahi[1,2], Thomas Jensen[1,3], Rasmus Ejlers Møgelberg[4], and Andrzej Wąsowski[4]

[1] University of Copenhagen, Denmark
[2] Paperflow, Denmark
[3] INRIA, France
[4] IT University of Copenhagen, Denmark

Abstract. Building a static analyser for a real language involves modeling of large domains capturing the many available data types. To scale domain design and support efficient development of project-specific analyzers, it is desirable to be able to build, extend, and change abstractions in a systematic and modular fashion. We present a framework for modular design of abstract domains for recursive types and higher-order functions, based on the theory of solving recursive domain equations. We show how to relate computable abstract domains to our framework, and illustrate the potential of the construction by modularizing a monolithic domain for regular tree grammars. A prototype implementation in the dependently typed functional language Agda shows how the theoretical solution can be used in practice to construct static analysers.

1 Introduction

It requires much work to construct a static analyzer using abstract interpretation. We must design suitable abstract domains for the properties we want to analyze, construct an abstract semantics that works with these domain, prove that this is sound with regards to the concrete semantics of our target programming language, and finally provide an implementation that reflects the developed theory and works efficiently in practice. A small change of the abstract domains to allow verifying new properties could have a large cascading effect w.r.t. the semantics, the proof of soundness and implementation. This is especially true for realistic programming language which have complex types, data structures and collections, and whose domains are often specified in a monolithic fashion.

There is therefore a need for compositional techniques for constructing abstract interpreters and proving their soundness, where multiple frameworks have been developed in recent years [14, 17, 16, 6]. These frameworks are advantageous since they allow changing parts of a static analysis in an isolated manner without completely changing the whole system. This makes it easier to continuously improve on static analyzers and exchange abstract domains for the target systems.

There is still a gap when it comes to compositionally analysing *recursive data structures*, which are present in many realistic programming languages. They

* Authors ordered alphabetically

are particularly challenging when combined with other domains, since recursive references can be present deeply inside data structures and nested inside complex domains like collections or functions. This paper proposes a systematic approach for designing and implementing abstract domains for complex types, based on the theory of solving recursive domain equations. We present a method for compositional construction of abstract domains that allows changing abstractions at the granularity of types. For instance changing the abstraction of shapes, without affecting the abstraction of integers or functions or vice versa.

The central theoretical contribution of this paper is to extend the construction of Galois connections for recursive data types. As far as we know, we are first to establish such a correspondence between concrete and abstract domains combinations that include recursive equations and higher-order functions. The solution is based on solving recursive domain equations over a category of complete lattices. Furthermore, we show how the construction naturally leads to a compositional implementation of concrete and abstract semantics for a recursive expression language (Sect. 6), in which the modularity of domains allows changing operations for select types without requiring rewriting the rest of semantics.

The general solution for abstracting recursive domains provides a modular precise mathematical formulation, but is not computable. We present two computable abstractions for recursive types: k-limited trees and typed regular tree grammars. k-limited trees allow us to show a property up to a limited depth k, while typed regular tree grammars allow us to infer a refined abstract data type. To see the latter, consider the negation normal form program in Fig. 1 (written in ML-like syntax). The program transforms abstract syntax of propositional formulas (e.g., $\neg(a \wedge (b \vee c))$) to equivalent ones where negations only appear in front of atoms (e.g., $\neg a \vee (\neg b \wedge \neg c)$). The typed regular tree grammar domain would *e.g.*, allow us to infer a grammar that precisely characterizes that the output of the program (nnf) only has negations in front of atoms as required:

$$\mathit{NFml} ::= \mathrm{Atom}(\mathsf{int}) \mid \mathrm{Neg}(\mathrm{Atom}(\mathsf{int})) \mid \mathrm{And}(\mathit{NFml}, \mathit{NFml}) \mid \mathrm{Or}(\mathit{NFml}, \mathit{NFml})$$

We show how to modularize the monolithic abstract domain containing these elements, dealing with the core challenges: representing syntactic recursion (e.g., over symbol NFml) while preserving typing, and compositionality of the lattice operations.

We show the correctness of the computable abstractions for recursive types by relating them to the general domain, thereby allowing reuse of proofs for most subcomponents (including base types, sum types and product types). The k-limited tree domain is fully modular and can be related using Galois connections to the general abstract domain $[\![A]\!]^\sharp \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} [\![A]\!]^\sharp_{\mathrm{K}}$, and by composition to the concrete power-set domain $\mathscr{P}[\![A]\!] \underset{\alpha'}{\overset{\gamma'}{\leftrightarrows}} [\![A]\!]^\sharp_{\mathrm{K}}$. The more complex regular tree grammar domain is related to general abstract domain via concretization.

The paper is structured as follows. We define a simply-typed lambda calculus and its set-based collecting semantics in Sect. 2 to guide the presentation of our paper. Sect. 3 summarizes relevant background on abstract interpretation and

```
type Fml = Atom(int) | Neg(Fml) | And(Fml, Fml) | Or(Fml, Fml)

nnf (f: Fml) = case f of
| Atom(i) -> Atom(i) | Neg(Atom(i)) -> Neg(Atom(i))
| Neg(Neg(f)) -> nnf f
| Neg(And(f1, f2)) -> Or(nnf (Neg f1), nnf (Neg f2))
| Neg(Or(f1, f2)) -> And(nnf (Neg f1), nnf (Neg f2))
| Or(f1, f2) -> Or(nnf f1, nnf f2)
| And(f1, f2) -> And(nnf f1, nnf f2)
```

Fig. 1: Negation Normal Form

Galois connections for unit type, sums, products and functions. Sect. 4 shows our solution for constructing Galois connections between concrete and abstract interpretation of recursive types, including the necessary proofs. Sect. 5 shows two computable compositional abstract domains for recursive types, and how to relate them to the general abstract domain. Sect. 7 presents related work and Sect. 8 concludes.

2 Expression Language and its Collecting Semantics

We highlight the strengths of our technique using a typed λ-calculus with recursion.

The set of types include integers (int), the unit type (unit), sum types ($A_1{+}A_2$), product types ($A_1{*}A_2$), function types ($A_1 \Rightarrow A_2$), and recursive types ($\mu X.A$). The grammar of types is given by:

$$A ::= \mathsf{int} \mid \mathsf{unit} \mid \mu X.B \mid A_1{*}A_2 \mid A_1{+}A_2 \mid A_1 \Rightarrow A_2$$
$$B ::= \mathsf{int} \mid \mathsf{unit} \mid B_1{*}B_2 \mid B_1{+}B_2 \mid A \Rightarrow B \mid X$$

In order to accommodate recursive types, we introduce a type variable X which means that we now have closed types (metavariable A) and open types (metavariable B). For simplicity of presentation, we only have one type variable X which effectively disallows nested recursive types—we believe our results extend to them as well. The grammar for open types (metavariable B) restricts recursive types to be strictly positive, meaning that the type variable can not appear to the left of an arrow. This restriction is imposed in order to have a set theoretic interpretation of both higher order and recursive types.

The expression language is intrinsically typed and contains usual operations for each type, variables, and a fixed-point combinator ($\mathsf{fix}\, x.e$). The $\mathsf{unfold}\ e$ and $\mathsf{fold}\ e$ expressions respectively allow accessing/abstracting away the underlying representation of expressions of recursive types. The typing judgment for expressions has form $\Gamma \vdash_{\mathsf{ex}} e : A$, ensuring that expression e has type A under some typing context Γ. The associated rules are standard and given in Fig. 2.

$$\mathsf{bool} = \mathsf{unit}{+}\mathsf{unit} \quad x_0 : A_0, \ldots, x_\mathrm{n} : A_\mathrm{n} \vdash_{\mathsf{ex}} x_i : A_i \quad \Gamma \vdash_{\mathsf{ex}} n : \mathsf{int} \quad \Gamma \vdash_{\mathsf{ex}} \mathsf{tt} : \mathsf{unit}$$

$$\frac{\Gamma \vdash_{\mathsf{ex}} e_1 : \mathsf{int} \quad \Gamma \vdash_{\mathsf{ex}} e_2 : \mathsf{int}}{\Gamma \vdash_{\mathsf{ex}} e_1 \odot e_2 : \mathsf{int}} \ (\odot \in \{+, -, \times\}) \qquad \frac{\Gamma \vdash_{\mathsf{ex}} e_1 : \mathsf{int} \quad \Gamma \vdash_{\mathsf{ex}} e_2 : \mathsf{int}}{\Gamma \vdash_{\mathsf{ex}} e_1 \leq e_2 : \mathsf{bool}}$$

$$\frac{\Gamma \vdash_{\mathsf{ex}} e_1 : A_1 \quad \Gamma \vdash_{\mathsf{ex}} e_2 : A_2}{\Gamma \vdash_{\mathsf{ex}} (e_1, e_2) : A_1 * A_2} \qquad \frac{\Gamma \vdash_{\mathsf{ex}} e : A_1 * A_2}{\Gamma \vdash_{\mathsf{ex}} \mathsf{fst}\ e : A_1} \qquad \frac{\Gamma \vdash_{\mathsf{ex}} e : A_1 * A_2}{\Gamma \vdash_{\mathsf{ex}} \mathsf{snd}\ e : A_2}$$

$$\frac{\Gamma \vdash_{\mathsf{ex}} e : A_1}{\Gamma \vdash_{\mathsf{ex}} \mathsf{inl}\ e : A_1 + A_2} \qquad \frac{\Gamma \vdash_{\mathsf{ex}} e : A_2}{\Gamma \vdash_{\mathsf{ex}} \mathsf{inr}\ e : A_1 + A_2}$$

$$\frac{\Gamma \vdash_{\mathsf{ex}} e : A_1 + A_2 \quad \Gamma, x : A_1 \vdash_{\mathsf{ex}} e_1 : A \quad \Gamma, y : A_2 \vdash_{\mathsf{ex}} e_2 : A}{\Gamma \vdash_{\mathsf{ex}} \mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}\ x \to e_1 \parallel \mathsf{inr}\ y \to e_2 : A}$$

$$\frac{\Gamma \vdash_{\mathsf{ex}} e : B\,[\mu X.B/X]}{\Gamma \vdash_{\mathsf{ex}} \mathsf{fold}\ e : \mu X.B} \qquad \frac{\Gamma \vdash_{\mathsf{ex}} e : \mu X.B}{\Gamma \vdash_{\mathsf{ex}} \mathsf{unfold}\ e : B\,[\mu X.B/X]} \qquad \frac{\Gamma, x : A \vdash_{\mathsf{ex}} e : A}{\Gamma \vdash_{\mathsf{ex}} \mathsf{fix}\ x.e : A}$$

$$\frac{\Gamma, x : A_1 \vdash_{\mathsf{ex}} e : A_2}{\Gamma \vdash_{\mathsf{ex}} \lambda x.e : A_1 \Rightarrow A_2} \qquad \frac{\Gamma \vdash_{\mathsf{ex}} e_1 : A_2 \Rightarrow A_1 \quad \Gamma \vdash_{\mathsf{ex}} e_2 : A_2}{\Gamma \vdash_{\mathsf{ex}} e_1\ e_2 : A_1}$$

Fig. 2: Typing Rules of the λ-calculus

The set-theoretic interpretation of types is the expected one, interpreting each syntactic construct into the corresponding semantic one. Because X is required to appear only strictly positively in B, the type can be interpreted as a functor on the category of sets $[\![B]\!] : \mathsf{Set} \to \mathsf{Set}$ defined by induction on B as follows

$$[\![\mathsf{int}]\!](X) = \mathbb{Z} \quad [\![\mathsf{unit}]\!](X) = \mathbb{1} \quad [\![X]\!](X) = X$$
$$[\![B_1 * B_2]\!](X) = [\![B_1]\!](X) \times [\![B_2]\!](X) \quad [\![B_1 + B_2]\!](X) = [\![B_1]\!](X) + [\![B_2]\!](X)$$
$$[\![A]\!](X) = [\![A]\!] \quad [\![A \Rightarrow B]\!](X) = [\![A]\!] \to ([\![B]\!](X))$$

where in the last case we assume that the closed type A has already been given an interpretation. For function types we can assume X does not appear in A (by strict positivity), so we can interpret it directly as a set. The only elaborate interpretation is the one for recursive types, which we will discuss in Sect. 4.

The collecting semantics for open terms (with context) is interpreted according to their types:

$$[\![\Gamma \vdash_{\mathsf{ex}} t : A]\!] : [\![\Gamma]\!] \to \mathscr{P}[\![A]\!]$$

where we interpret contexts as products of sets of values:

$$[\![x_1 : B_1, \ldots, x_\mathrm{n} : B_\mathrm{n}]\!] = \prod_i \mathscr{P}[\![B_i]\!]$$

There are alternative choices [11] for the collecting semantics domain depending on the target abstract domains, e.g. $\mathscr{P}(\prod_i [\![B_i]\!]) \to \mathscr{P}([\![A]\!])$ or $\mathscr{P}(\prod_i [\![B_i]\!] \to [\![A]\!])$. We chose ours because it is the simplest one that suffices for our compositional domain as proven in Sect. 4.

The collecting semantics provides a concrete interpretation of a term $\vdash_{\mathsf{ex}} e : A$ as the set of possible concrete values $v \in \mathscr{P}[\![A]\!]$ that the term can evaluate to. In our case, we first provide a compositional interpretation of program types into sets, so $[\![A]\!] : \mathsf{Set}$, and then present how to specify collecting semantics for terms in our language, including the challenging case of higher-order functions and fixed-points. The collecting semantics is particularly useful when relating to our abstract domains $[\![A]\!]^\sharp$ which are complete lattices, since power sets $\mathscr{P}[\![A]\!]$ form complete lattices themselves. This makes it possible to compositionally specify Galois connection and show soundness of the abstract interpretation framework, which we do in sections 3 and 4.

We define most cases straightforwardly, ensuring monotonicity of mapping w.r.t the context. We omit the explicit typing of syntax for readability of the semantics. We write $\mathscr{P}(f) : \mathscr{P}([\![A_1]\!] \times \cdots \times [\![A_\mathrm{n}]\!]) \to \mathscr{P}[\![B]\!]$ to lift function $f : [\![A_1]\!] \times \cdots \times [\![A_\mathrm{n}]\!] \to [\![B]\!]$ to the power set domain, mimicking notation from category theory.

$$[\![n]\!]\rho = \{\mathrm{n}\} \quad [\![x]\!]\rho = \rho(x) \quad [\![\mathsf{tt}]\!]\rho = \{\star\}$$
$$[\![e_1 \odot e_2]\!]\rho = \mathscr{P}(\odot)([\![e_1]\!]\rho, [\![e_2]\!]\rho)$$
$$[\![e_1 \leq e_2]\!]\rho = \mathscr{P}(\leq)([\![e_1]\!]\rho, [\![e_2]\!]\rho) \quad [\![(e_1, e_2)]\!]\rho = \mathscr{P}(,)([\![e_1]\!]\rho, [\![e_2]\!]\rho)$$
$$[\![\mathsf{fst}\ e]\!]\rho = \mathscr{P}(\pi_1)([\![e]\!]\rho) \quad [\![\mathsf{snd}\ e]\!]\rho = \mathscr{P}(\pi_2)([\![e]\!]\rho)$$
$$[\![\mathsf{inl}\ e]\!]\rho = \mathscr{P}(\imath_1)([\![e]\!]\rho) \quad [\![\mathsf{inr}\ e]\!]\rho = \mathscr{P}(\imath_2)([\![e]\!]\rho)$$
$$[\![e_1\ e_2]\!]\rho = \{f(a) | f \in [\![e_1]\!]\rho, a \in [\![e_2]\!]\rho\}$$
$$[\![\mathsf{case}\ e\ \mathsf{of}\ \mathsf{inl}\ x \to e_1 \parallel \mathsf{inr}\ y \to e_2]\!]\rho = \bigcup_{i \in \{1,2\}} [\![e_i]\!]\rho\,[x \mapsto \{v | \imath_i v \in [\![e]\!]\rho\}]$$

The interpretation for functions is more intricate. The outcome of analysing the body of a function for a given argument is a set of results, and this has to be lifted to a set of functions. To do this, we define the (left-invertible) mapping $\delta : ([\![A]\!] \to \mathscr{P}[\![B]\!]) \to \mathscr{P}([\![A]\!] \to [\![B]\!])$:

$$\delta(f) = \{g : [\![A]\!] \to [\![B]\!] | \forall a \in [\![A]\!].g(a) \in f(a)\}$$

Then we can interpret functions using δ into our target set of functions:

$$\begin{aligned}[\![\rho \vdash_{\mathsf{ex}} \lambda x.e : A \to B]\!]\rho &= \delta(\lambda a.[\![e]\!]\rho[x \mapsto \{a\}]) \\ &= \{f \in [\![A]\!] \to [\![B]\!] \mid \forall a \in [\![A]\!].f(a) \in [\![e]\!]\rho[x \mapsto \{a\}]\}\end{aligned}$$

Finally, we can then interpret fixed-points by relying on the power set domain $\mathscr{P}([\![A]\!])$ forming a complete lattice:

$$[\![\rho \vdash_{\mathsf{ex}} \mathsf{fix}\ x.e : A]\!]\rho = \mathsf{lfp}(\lambda X.[\![e]\!]\rho[x \mapsto X])$$

where the least fixed-point on complete lattices $\mathsf{lfp}(f) = \bigcap\{a \mid f(a) \subseteq a\}$ is given by the Knaster-Tarski Fixed-point Theorem [24] and Theorem 1. Now, that we have provided a compositional collecting semantics for all our language constructs, we can proceed with the abstract domains.

Theorem 1. *If* $\rho_1 \subseteq \rho_2$ *then* $[\![e]\!]\rho_1 \subseteq [\![e]\!]\rho_2$

3 Abstract interpretation and Galois connections

Abstract interpretation provides a theory for relating a concrete set of values for a type $\mathscr{P}[\![A]\!]$ to an abstract (usually complete) lattice of elements for the same type $[\![A]\!]^\sharp$. The relation is performed using Galois connections [10] $\mathscr{P}[\![A]\!] \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} [\![A]\!]^\sharp$, which uses two monotone maps $\alpha : \mathscr{P}[\![A]\!] \to [\![A]\!]^\sharp$ and $\gamma : [\![A]\!]^\sharp \to \mathscr{P}[\![A]\!]$ to map between the two domains, so that the following relation is preserved $a \subseteq \gamma(a^\sharp) \iff \alpha(a) \sqsubseteq a^\sharp$. The existence of a Galois connection formalises the way in which an abstract property describes a set of concrete values, provides a notion of "best approximation" and even suggests a design methodology for deriving program analysers in a systematic way by composing the semantic function with an abstraction function α [9, 18]

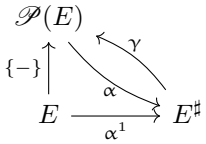
We are interested in abstract interpretation in which abstract domains are defined as compositional interpretations of types, where the semantics of a type is composed of the types it contains. E.g., the semantics of $A*B$ is composed of the semantics of A and B. For base types and first-order algebraic types there is a well-established theory for relating the concrete set-theoretic semantics $[\![A]\!]$ and the corresponding abstract interpretation to complete lattices $[\![A]\!]^\sharp$ in a compositional fashion, using Galois connections $\mathscr{P}[\![A]\!] \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} [\![A]\!]^\sharp$. This design is what makes it possible to exchange abstractions at the granularity of types, while preserving the structure of other parts of the system.

The following theorem is useful for constructing Galois connections, by using a map $\alpha^1 : [\![A]\!] \to [\![A]\!]^\sharp$ that directly abstracts concrete values [20, p. 237].

Proposition 1. *Let E be a set and $E^\sharp$ a complete lattice. There is a bijective correspondence between maps $\alpha^1 : E \to E^\sharp$ and Galois connections $\mathscr{P}(E) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} E^\sharp$. The correspondence maps a Galois connection to α^1 defined as $\alpha^1(x) = \alpha(\{x\})$ and a map α^1 to the Galois connection:*

$$\alpha(X \subseteq E) = \bigsqcup_{x \in X} \alpha^1(x) \qquad \gamma(e^\sharp) = \left\{ x \in E \;\middle|\; \alpha^1(x) \sqsubseteq e^\sharp \right\}$$

The correspondence can be summarized in the following diagram.



The systematic construction of Galois connections for base types, sums, and products is standard [20, sect. 4.4]—we recall them since they are needed for recursive types (Sect. 4).

Unit type. We abstract the unit type as the 2-element lattice $[\![\mathsf{unit}]\!]^\sharp = \Sigma = \{\bot, \top\}$ where $\bot \leq \top$. The induced Galois connection is the isomorphism $\mathscr{P}(\mathbb{1}) \cong \Sigma$ mapping $\emptyset$ to $\bot$ and $\{\star\}$ to $\top$. Similarly, we abstract tt as $\top$, $[\![\mathsf{tt}]\!]^\sharp \rho^\sharp = \top$.

Sums. The abstract interpretation of a sum type is the product of lattices (with the pointwise ordering):$[\![A_1 + A_2]\!]^\sharp = [\![A_1]\!]^\sharp \times [\![A_2]\!]^\sharp$. The intuition is that each component of a pair (a_1, a_2) in the product $[\![A_1]\!]^\sharp \times [\![A_2]\!]^\sharp$ describes a property of the sum value *if* the value belongs to the corresponding summand; for concrete values $\imath_1(v)$ (corr. $\imath_2(v)$) we will have that v satisfies the property a_1 (corr. a_2). The element abstraction map is defined as:

$$\alpha^1_{A_1+A_2}(\imath_1(x)) = (\alpha^1_{A_1}(x), \bot) \qquad \alpha^1_{A_1+A_2}(\imath_2(y)) = (\bot, \alpha^1_{A_2}(y))$$

where $\imath_1$ and $\imath_2$ are the injections into the disjoint sum.

Products. The simplest useful abstraction of pairs is by using the smash product of abstract domains ($A^\sharp \otimes B^\sharp$). The smash product is a product that disallows pairs where only one component is the bottom element:

$$[\![A_1 * A_2]\!]^\sharp = A^\sharp \otimes B^\sharp = \{(a, b) | a = \bot \iff b = \bot\}$$

We will implicitly convert a pair which has a single $\bot$ component, to one where both components are $\bot$. Using this, the element abstraction map is defined as

$$\alpha^1_{A_1 * A_2}(x, y) = (\alpha^1_{A_1}(x), \alpha^1_{A_2}(x))$$

Other abstract domain constructions for products are based on ordinary Cartesian products or tensor products [19] which keep relational information between the two components. We conjecture that the theory extends to these other types of products but we leave the formal verification of this for further work.

Function types. We abstractly interpret function types as the set of monotone functions:

$$[\![A_1 \Rightarrow A_2]\!]^\sharp = \{f : [\![A_1]\!]^\sharp \to [\![A_2]\!]^\sharp \mid \forall x, y. x \leq y \Longrightarrow f(x) \leq f(y)\}$$

By induction, we can assume the Galois connection is given for the types A_1 and A_2 and define,

$$\alpha^1_{A_1 \Rightarrow A_2} : [\![A_1 \Rightarrow A_2]\!] \to [\![A_1 \Rightarrow A_2]\!]^\sharp$$

to map $f : [\![A_1]\!] \to [\![A_2]\!]$ to the composition

$$\alpha_{A_2} \circ \mathscr{P}(f) \circ \gamma_{A_1}$$

Note that $\mathscr{P}(f)$ is monotone, and so the composition is monotone, but since γ_{A_1} is not necessarily continuous, neither is the composition. This is one reason $[\![A_1 \Rightarrow A_2]\!]^\sharp$ needs to be the set of monotone, rather than continuous maps. Monotone functions over complete lattices form themselves a complete lattice.

4 Recursive types

We now extend the theory of Section 3 with recursive types. We shall define abstract domains for recursive types as solutions to recursive domain equations. Categorically, we rely on *initial algebras* to describe inductive types. An initial algebra for some functor $F : \mathsf{Set} \to \mathsf{Set}$ consists of a set X (the carrier) and a map $f : FX \to X$ such that for any other map $g : FY \to Y$, there exists a unique map $h : X \to Y$ making the following diagram commute:

$$\begin{array}{ccc} FX & \xrightarrow{Fh} & FY \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle g} \\ X & \xrightarrow{h} & Y \end{array}$$

For example, the initial algebra for the functor $F(X) = 1 + \mathbb{Z} \times X$ is exactly the set of lists of integers, and the initiality property described above captures exactly the induction principle for lists. Initial algebras generalise directly from Set to other categories, see e.g., Awodey [4, Chp 10.5] for more details.

We rely on strict positivity to interpret $\mu X.B$, since the corresponding functor $[\![B]\!](-)$ is known to have an initial algebra [1]. Formally, $\mu X.B$ is interpreted as the carrier of the initial algebra for $[\![B]\!](-)$.

4.1 Recursively defined lattices

To define the abstract interpretation of recursive types, open types must be interpreted as functors on a category of complete lattices. For this, let cLat denote the category of complete lattices and continuous maps, i.e., maps preserving all least upper bounds. Recall that this implies that maps are monotone and preserve the bottom element.

Given two complete lattices $X^\sharp$, $Y^\sharp$, the set of maps (hom-set) between them $\mathsf{cLat}(X^\sharp, Y^\sharp)$ is itself a complete lattice under the pointwise ordering, i.e., $f \sqsubseteq g$ if $f(x) \sqsubseteq g(x)$ for all x. A functor $F : \mathsf{cLat} \to \mathsf{cLat}$ is *locally continuous* if the lifting of maps $f \in \mathsf{cLat}(X^\sharp, Y^\sharp)$ to work at the functor level $Ff \in \mathsf{cLat}(F(X^\sharp), F(Y^\sharp))$ is itself continuous, i.e., $F(\bigsqcup fs) = \bigsqcup_{f \in fs} F(f)$.

Theorem 2. *Any locally continuous functor $F : \mathsf{cLat} \to \mathsf{cLat}$ has a fixed point $\mathsf{Fix}\, F \cong F(\mathsf{Fix}\, F)$.*

Proof. The proof is completely standard [2, 23], but we recall the construction of the fixed point for future reference. Let $\mathbb{1} = \{\bot\}$ be the singleton lattice. There are continuous maps $e_0 : \mathbb{1} \to F\mathbb{1}$ (mapping $\bot$ to the bottom element of $F(\mathbb{1})$) and $p_0 : F\mathbb{1} \to \mathbb{1}$ forming an embedding-projection pair, i.e., satisfying $p_0 \circ e_0 = \mathsf{id}$ and $e_0 \circ p_0 \sqsubseteq \mathsf{id}$. Defining $p_n = F^n p_0$ and $e_n = F^n e_0$ gives an embedding-projection pair from $F^n(\mathbb{1})$ to $F^{n+1}(\mathbb{1})$. The fixed point is defined as the limit of the chain of p_n maps, i.e., $\mathsf{Fix}\, F = \{(x_1, x_2, \dots) \mid \forall n. x_n \in F^n(\mathbb{1}), x_n = p_n(x_{n+1})\}$

To define the lattice interpretation of types, it thus suffices to construct an interpretation of each open type expression as a locally continuous functor $\mathsf{cLat} \to \mathsf{cLat}$. The interpretation is defined using the constructions of Section 3:

$$[\![X]\!]^\sharp(X^\sharp) = X^\sharp \qquad [\![B_1 * B_2]\!]^\sharp(X^\sharp) = [\![B_1]\!]^\sharp(X^\sharp) \otimes [\![B_2]\!]^\sharp(X^\sharp)$$
$$[\![B_1 + B_2]\!]^\sharp(X^\sharp) = [\![B_1]\!]^\sharp(X^\sharp) \times [\![B_2]\!]^\sharp(X^\sharp)$$
$$[\![A \Rightarrow B]\!]^\sharp(X^\sharp) = [\![A]\!]^\sharp \underset{\text{mono}}{\to} ([\![B]\!]^\sharp(X^\sharp))$$

Here, the notation $\underset{\text{mono}}{\to}$ refers to the lattice of monotone functions.

4.2 Defining the Galois connection

Having defined the interpretation of each type A as a set $[\![A]\!]$ and a lattice $[\![A]\!]^\sharp$ respectively, we must now construct a Galois connection from $\mathscr{P}[\![A]\!]$ to $[\![A]\!]^\sharp$. This will be defined by induction on the structure of A and so must also take open types into account. For this, we define for each open type B and each complete lattice $X^\sharp$ a map

$$\alpha^1_B(X^\sharp) : [\![B]\!](X^\sharp) \to [\![B]\!]^\sharp(X^\sharp)$$

where the codomain is the underlying set of the complete lattice $[\![A]\!]^\sharp(X^\sharp)$ (essentially forgetting the lattice structure). For most type constructors we just show the case of the open type, since the closed case is essentially the same. The basic cases are as follows:

$$\alpha^1_X(X^\sharp)(x) = x \qquad \alpha^1_{B_1 * B_2}(X^\sharp)(x, y) = (\alpha^1_{B_1}(X^\sharp)(x), \alpha^1_{B_2}(X^\sharp)(y))$$
$$\alpha^1_{B_1+B_2}(X^\sharp)(\imath_1(x)) = (\alpha^1_{B_1}(X^\sharp)(x), \bot) \qquad \alpha^1_{B_1+B_2}(X^\sharp)(\imath_2(x)) = (\bot, \alpha^1_{B_2}(X^\sharp)(x))$$

For function type $A \Rightarrow B$, by induction we have

$$\alpha^1_A : [\![A]\!] \to [\![A]\!]^\sharp \qquad \alpha^1_B(X^\sharp) : [\![B]\!](X^\sharp) \to [\![B]\!]^\sharp(X^\sharp)$$

(since A is a closed type). These induce

$$\gamma_A : [\![A]\!]^\sharp \to \mathscr{P}([\![A]\!]) \qquad \alpha_B(X^\sharp) : \mathscr{P}([\![B]\!](X^\sharp)) \to [\![B]\!]^\sharp(X^\sharp)$$

by Proposition 1. Thus, we can define

$$\alpha^1_{A \Rightarrow B}(X^\sharp) : ([\![A]\!] \to [\![B]\!](X^\sharp)) \to ([\![A]\!]^\sharp \underset{\text{mono}}{\to} [\![B]\!]^\sharp(X^\sharp))$$

to map g to the composition $\alpha_B(X^\sharp) \circ \mathscr{P}(g) \circ \gamma_A$.

In the case of recursive types, recall that $[\![\mu X.B]\!]$ is defined as the initial algebra for the functor $[\![B]\!] : \mathsf{Set} \to \mathsf{Set}$. To define $\alpha^1_{\mu X.B}$ we therefore define an algebra structure

$$[\![B]\!]([\![\mu X.B]\!]^\sharp) \to [\![\mu X.B]\!]^\sharp$$

for $[\![\mu X.B]\!]^\sharp$ as the composite of

$$\alpha^1_B([\![\mu X.B]\!]^\sharp) : [\![B]\!]([\![\mu X.B]\!]^\sharp) \to [\![B]\!]^\sharp([\![\mu X.B]\!]^\sharp)$$

and the isomorphism

$$\mathsf{fold}^\sharp : [\![B]\!]^\sharp([\![\mu X.B]\!]^\sharp) \to [\![\mu X.B]\!]^\sharp$$

Thus, we define $\alpha^1_{\mu X.B}$ to be the unique map that makes the diagram commute:

$$\begin{array}{ccc} [\![B]\!]([\![\mu X.B]\!]) & \xrightarrow{[\![B]\!](\alpha^1_{\mu X.B})} & [\![B]\!]([\![\mu X.B]\!]^\sharp) \\ \downarrow & & \downarrow \\ [\![\mu X.B]\!] & \xrightarrow{\alpha^1_{\mu X.B}} & [\![\mu X.B]\!]^\sharp \end{array}$$

The vertical map on the left is the algebra structure for $[\![\mu X.B]\!]$ and the map $\alpha^1_{\mu X.B}$ exists uniquely by the initial algebra property.

Note that this construction crucially uses that the set-interpretation of recursive types is as initial algebras[5]. On the other hand, we do not use anything specific about the lattice interpretation, and in fact, the construction of the Galois connection simply requires a map

$$[\![B]\!]^\sharp([\![\mu X.B]\!]^\sharp) \to [\![\mu X.B]\!]^\sharp$$

not necessarily an isomorphism. We will use this fact in Sect. 5, and for this reason we state the main theorem in a more general setting.

Theorem 3. *Suppose the abstract interpretation of recursive types*

$$\mathsf{fold}^\sharp : [\![B\,[\mu X.B/X]]\!]^\sharp \to [\![\mu X.B]\!]^\sharp \qquad \mathsf{unfold}^\sharp : [\![\mu X.B]\!]^\sharp \to [\![B\,[\mu X.B/X]]\!]^\sharp$$

satisfies the equations

$$\mathsf{fold}^\sharp \circ \mathsf{unfold}^\sharp = \mathsf{id} \qquad \mathsf{unfold}^\sharp \circ \mathsf{fold}^\sharp \sqsupseteq \mathsf{id}$$

If $\vdash_{\mathsf{ex}} t : A$ *then* $[\![t]\!] \subseteq \gamma([\![t]\!]^\sharp)$*, or equivalently* $\alpha([\![t]\!]) \sqsubseteq [\![t]\!]^\sharp$

Theorem 3 is proved by induction on typing judgements, and must therefore be extended to open terms. This is done in the following lemma, which uses an extension of the maps γ and α to contexts defined pointwise:

Lemma 1. *Suppose* $\Gamma \vdash_{\mathsf{ex}} e : A$*, and that* $\rho \in [\![\Gamma]\!]$ *and* $\rho^\sharp \in [\![\Gamma]\!]^\sharp$*, are such that* $\rho \subseteq \gamma(\rho^\sharp)$*. If the conditions on the abstract interpretation of recursive types of Theorem 3 are satisfied, then* $[\![t]\!]\rho \subseteq \gamma([\![t]\!]^\sharp\rho^\sharp)$ *or, equivalently,* $\alpha([\![t]\!]\rho) \subseteq [\![t]\!]^\sharp\rho^\sharp$

Theorem 3 obviously follows from this as a special case. Before proving Lemma 1 we need a few lemmas.

[5] The construction would not work if we used final coalgebras

Lemma 2. *If B is an open type and A a closed one, then $[\![B]\!]([\![A]\!]) = [\![B\,[A/X]]\!]$ and $[\![B]\!]^\sharp([\![A]\!]^\sharp) = [\![B\,[A/X]]\!]^\sharp$*

Proof. Easy induction on B, which we omit.

Lemma 3. *For any open type B and closed type A the following equality holds*

$$\alpha^1_B([\![A]\!]^\sharp) \circ [\![B]\!](\alpha^1_A) = \alpha^1_{B[A/X]}$$

In diagram style, this is

$$\begin{array}{ccccc} [\![B\,[A/X]]\!] & \xrightarrow{\mathsf{id}} & [\![B]\!]([\![A]\!]) & \xrightarrow{[\![B]\!](\alpha^1_A)} & [\![B]\!]([\![A]\!]^\sharp) \\ \downarrow{\scriptstyle \alpha^1_{B[A/X]}} & & & & \downarrow{\scriptstyle \alpha^1_B([\![A]\!]^\sharp)} \\ [\![B\,[A/X]]\!]^\sharp & & \xrightarrow{\qquad\qquad\mathsf{id}\qquad\qquad} & & [\![B]\!]^\sharp([\![A]\!]^\sharp) \end{array}$$

Proof. Induction on B (Appendix A).

Lemma 4. *If B is an open type, then the following diagram commutes*

$$\begin{array}{ccc} \mathscr{P}([\![B\,[\mu X.B/X]]\!]) & \xrightarrow{\alpha_{B[\mu X.B/X]}} & [\![B\,[\mu X.B/X]]\!]^\sharp \\ \downarrow{\scriptstyle \mathscr{P}(\mathsf{fold})} & & \downarrow{\scriptstyle \mathsf{fold}^\sharp} \\ \mathscr{P}([\![\mu X.B]\!]) & \xrightarrow{\alpha_{\mu X.B}} & [\![\mu X.B]\!]^\sharp \end{array}$$

Proof. It suffices to show that $\alpha^1_{\mu X.B} \circ \mathsf{fold} = \mathsf{fold}^\sharp \circ \alpha^1_{B[\mu X.B/X]}$ since the two compositions in the diagram are the unique extensions of the two sides of this equation to continuous maps. By definition of $\alpha^1_{\mu X.B}$ we get

$$\alpha^1_{\mu X.B} \circ \mathsf{fold} = \mathsf{fold}^\sharp \circ \alpha^1_B([\![\mu X.B]\!]^\sharp) \circ [\![B]\!](\alpha^1_{\mu X.B})$$

By Lemma 3 the right hand side of this is equal to $\mathsf{fold}^\sharp \circ \alpha^1_{B[\mu X.B/X]}$ which concludes the proof.

We can now prove Lemma 1.

Proof (Lemma 1). By induction on typing derivation $\Gamma \vdash_{\mathsf{ex}} e : A$. We show the interesting cases for recursive types and the rest are in Appendix A.
Case $t = \mathsf{fold}\ u : \mu X.A$,

$$\alpha([\![\mathsf{fold}\ u]\!]\rho) = \alpha(\mathscr{P}(\mathsf{fold})([\![u]\!]\rho)) = \mathsf{fold}^\sharp(\alpha([\![u]\!]\rho)) \sqsubseteq \mathsf{fold}^\sharp([\![u]\!]^\sharp\rho^\sharp) = [\![\mathsf{fold}\ u]\!]^\sharp\rho^\sharp$$

– Case $t = \mathsf{unfold}\ u : A\,[\mu X.A/X]$
  First note that

$$\alpha \circ \mathscr{P}(\mathsf{unfold}) \sqsubseteq \mathsf{unfold}^\sharp \circ \alpha$$

  since

$$\begin{aligned}\alpha \circ \mathscr{P}(\mathsf{unfold}) &\sqsubseteq \mathsf{unfold}^\sharp \circ \mathsf{fold}^\sharp \circ \alpha \circ \mathscr{P}(\mathsf{unfold}) \\ &= \mathsf{unfold}^\sharp \circ \alpha \circ \mathscr{P}(\mathsf{fold} \circ \mathsf{unfold}) = \mathsf{unfold}^\sharp \circ \alpha\end{aligned}$$

  and so

$$\begin{aligned}\alpha([\![\mathsf{unfold}\ u]\!]\rho) &= \alpha(\mathscr{P}(\mathsf{unfold})([\![u]\!]\rho)) \\ &\sqsubseteq \mathsf{unfold}^\sharp(\alpha([\![u]\!]\rho)) \overset{\text{by IH}}{\sqsubseteq} \mathsf{unfold}^\sharp([\![u]\!]^\sharp\rho^\sharp) = [\![\mathsf{unfold}\ u]\!]^\sharp\rho^\sharp\end{aligned}$$

We can finally provide a compositional interpretation for program expressions manipulating recursive types, based on the corresponding concrete and abstract semantic operations for recursive types:

$$\begin{aligned}[\![\mathsf{fold}\ e]\!]\rho &= \mathscr{P}(\mathsf{fold})([\![e]\!]\rho) & [\![\mathsf{unfold}\ e]\!]\rho &= \mathscr{P}(\mathsf{unfold})([\![e]\!]\rho) \\ [\![\mathsf{fold}\ e]\!]^\sharp\rho^\sharp &= \mathsf{fold}^\sharp([\![e]\!]^\sharp\rho^\sharp) & [\![\mathsf{unfold}\ e]\!]^\sharp\rho^\sharp &= \mathsf{unfold}^\sharp([\![e]\!]^\sharp\rho^\sharp)\end{aligned}$$

We present an elaborate example illustrating how the constructs can be explicitly instantiated for lists of integers in Sect. 4.3. This is primarily to provide a more concrete formal intuition about the solutions to the fixed-points, since our theory works with all inductive types in general.

4.3 Example: integer lists

We illustrate the theoretical construction of Galois connections for recursive types on the simplest case of such types: lists of integers. Recall that the recursive type of an integer list is $\mu X.\mathsf{unit}{+}\mathsf{int}{*}X$, where the first component of the sum represents the empty list and the second component represents a pair of an integer element—representing the head element—and the rest of the list (tail).

In the standard semantics, the interpretation of this type is the initial algebra of the functor $[\![\mathsf{unit}{+}\mathsf{int}{*}X]\!](X) = 1 + \mathbb{Z} \times X$, which is simply the set of lists of integers. For the abstract semantics we interpret integers using the standard Sign abstraction, which abstracts a set of integers by the sign of elements ($+$, $-$ or 0) it contains (if all elements have the same sign), or otherwise the bottom $\bot$ (representing the empty set) or $\top$ (abstracting of sets with elements of mixed signs).

Then, $[\![\mu X.\mathsf{unit}{+}\mathsf{int}{*}X]\!]^\sharp$ is the fixed point of the functor

$$[\![\mathsf{unit}{+}\mathsf{int}{*}X]\!]^\sharp(X) = \Sigma \times (\mathsf{Sign} \otimes X)$$

The fixed point is constructed as in the proof of Theorem 2, i.e., as a limit of a chain obtained by applying F countably many times to the singleton lattice $\mathbb{1}$.

We start by computing a few iterations.

$$
\begin{aligned}
F(\mathbb{1}) &= \Sigma \times (\mathsf{Sign} \otimes \mathbb{1}) \cong \Sigma \\
F^2(\mathbb{1}) &\cong \Sigma \times (\mathsf{Sign} \otimes \Sigma) \cong \{(b, m_0, b_0) \in \Sigma \times \mathsf{Sign} \times \Sigma \mid m_0 = \bot \iff b_0 = \bot\} \\
F^3(\mathbb{1}) &\cong \Sigma \times (\mathsf{Sign} \otimes (\Sigma \times (\mathsf{Sign} \otimes \Sigma))) \\
&= \left\{ (b, m_0, b_0, m_1, b_1) \in \Sigma \times (\mathsf{Sign} \times \Sigma)^2 \middle| \begin{array}{c} (m_0 = \bot \iff (b_0 = \bot \wedge m_1 = \bot)) \wedge \\ (m_1 = \bot \iff b_1 = \bot) \end{array} \right\}
\end{aligned}
$$

In all cases, the map $F^{n+1}(\mathbb{1}) \to F^n(\mathbb{1})$ forgets the last two elements. In general

$$
\begin{aligned}
F^n(\mathbb{1}) \cong \{(b, (m_i, b_i)_i) \in \Sigma \times (\mathsf{Sign} \times \Sigma)^{n-1} \mid m_{n-1} = \bot \iff b_{n-1} = \bot \\
\forall i < n. (m_i = \bot \iff \forall k \geq 0. (b_{i+k} = \bot \wedge m_{i+k+1} = \bot))\}
\end{aligned}
$$

and the limit becomes

$$
\begin{aligned}
[\![\mu X.\mathsf{unit} + \mathsf{int} * X]\!]^\sharp \cong \{(b, (m_i, b_i)_i) \in \Sigma \times (\mathsf{Sign} \times \Sigma)^{\mathbb{N}} \mid \\
\forall i. (m_i = \bot \iff \forall k. (b_{i+k} = \bot \wedge m_{i+k+1} = \bot))\}
\end{aligned}
$$

The algebra structure on the underlying set of this lattice has type

$$1 + \mathbb{Z} \times ([\![\mu X.\mathsf{unit} + \mathsf{int} * X]\!]^\sharp) \to [\![\mu X.\mathsf{unit} + \mathsf{int} * X]\!]^\sharp$$

and maps $\imath_1(\star)$ (where $\star$ is the unique element in 1) to

$$(\top, (\bot, \bot)_n)$$

and an element $\imath_2(m, (b, (m_i, b_i)_i))$ to $(\bot, (m'_i, b'_i)_i)$ where

$$(m'_i, b'_i) = \begin{cases} (m, b) & i = 0 \\ (m_{i-1}, b_{i-1}) & i > 0 \end{cases}$$

The map

$$\alpha^1_{\mu X.\mathsf{unit} + \mathsf{int} * X} : [\![\mu X.\mathsf{unit} + \mathsf{int} * X]\!] \to [\![\mu X.\mathsf{unit} + \mathsf{int} * X]\!]^\sharp$$

which is defined using the initial algebra property of the set of lists thus acts as follows

$$
\begin{aligned}
\alpha^1_{\mu X.\mathsf{unit} + \mathsf{int} * X}([]) &= (\top, (\bot, \bot)_n) \\
\alpha^1_{\mu X.\mathsf{unit} + \mathsf{int} * X}([x_0, \ldots, x_n]) &= (\bot, (\mathsf{sign}(x_0), \bot), \ldots, (\mathsf{sign}(x_n), \top), (\bot, \bot), \ldots)
\end{aligned}
$$

The abstraction map

$$\alpha_{\mu X.\mathsf{unit} + \mathsf{int} * X} : \mathscr{P}[\![\mu X.\mathsf{unit} + \mathsf{int} * X]\!] \to [\![\mu X.\mathsf{unit} + \mathsf{int} * X]\!]^\sharp$$

maps a set X to the least upper bound of $\alpha^1_{\mu X.\mathsf{unit} + \mathsf{int} * X}$ applied to the elements of the set. For example, if

$$A = \left\{ xs \in [\![\mu X.\mathsf{unit} + \mathsf{int} * X]\!] \middle| \begin{array}{c} \exists n. \mathsf{length}(xs) = 2n. \\ \forall i < n. \mathsf{sign}(xs[2i]) = +, \mathsf{sign}(xs[2i+1]) = - \end{array} \right\}$$

is the set whose elements are lists of even length with alternating sign (starting with positive), then

$$\alpha_{\mu X.\mathsf{unit}+\mathsf{int}*X}(A) = (\top, (+, \bot), (-, \top), (+, \bot), (-, \top), \ldots)$$

is the corresponding abstraction. Concretely, we have $\bot$ every second time (so $b_{2i} = \bot$ for all i) to ensure that the length of the list is even.

The concretization map

$$\gamma_{\mu X.\mathsf{unit}+\mathsf{int}*X} : [\![\mu X.\mathsf{unit}+\mathsf{int}*X]\!]^\sharp \to \mathscr{P}[\![\mu X.\mathsf{unit}+\mathsf{int}*X]\!]$$

maps an abstract element to the set of all lists whose abstraction (by $\alpha_{\mu X.\mathsf{unit}+\mathsf{int}*X}$) are below it. For example, it maps $\alpha_{\mu X.\mathsf{unit}+\mathsf{int}*X}(A)$ back to A.

Another example is the top element of the abstract domain which is:

$$\top_{\mu X.\mathsf{unit}+\mathsf{int}*X} = (\top_\Sigma, (\top_{\mathsf{Sign}}, \top_\Sigma) \ldots)$$

It basically, uses the top element of each constituting abstract domain, while having the infinite form abstracting over all sizes of concrete lists, as required by the limit.

Limitation of the compositional solution A limitation of the compositional abstract solution for recursive types is that it cannot precisely capture constraints across the recursive structure. E.g., there is no precise abstract domain that characterizes lists where the elements are sorted. Providing a solution that can capture richer constraints is future work.

5 Computable Abstractions

The general abstraction of recursive types (Theorem 2) contains elements with a non-finitary structure, which makes it hard to use directly for terminating analyses. In this section, we show how to construct two computable abstract domains for recursive types: k-limited trees and typed regular tree expressions. We show how these can be constructed modularly and then easily related to the concrete powerset domain through the general abstract domain.

5.1 *k*-limited Trees

We can get a computable analysis from our general abstract domain, by only considering the subset of prefixes of recursive structures up to some fixed depth k. This subset of finitary elements is called k-limited trees. Formally, we define the k-limited trees abstract domain as follows

$$[\![\mu X.B]\!]^\sharp_{\mathrm{K}} = ([\![B]\!]^\sharp)^k(\Sigma)$$

where $[\![B]\!]^\sharp : \mathsf{cLat} \to \mathsf{cLat}$ is the abstract interpretation of B from Sect. 4.2.

We define $\mathsf{fold}_k^\sharp = (\llbracket B \rrbracket^\sharp)^k(!)$ where $! : \llbracket B \rrbracket^\sharp(X^\sharp) \to \Sigma$ is a morphism that forgets additional structure in $X^\sharp$ by mapping non-bottom elements to the top element and $\mathsf{unfold}_k^\sharp = (\llbracket B \rrbracket^\sharp)^k(\mathsf{inj})$ where $\mathsf{inj} : \Sigma \to \llbracket B \rrbracket^\sharp(\Sigma)$ is a morphism that maps bottom/top to bottom/top respectively. This interpretation of $\mathsf{fold}_k^\sharp$ and $\mathsf{unfold}_k^\sharp$ is suitable: only $\mathsf{fold}_k^\sharp$ introduces over-approximation, which is consistent with the conditions in Theorem 3. It also makes it clear why k is limiting: we can only unfold k times before losing information. The Galois connection of Section 4.2 extends to give a Galois connection from $\mathscr{P}(\llbracket \mu X.A \rrbracket)$ and $\llbracket \mu X.A \rrbracket^\sharp_{\mathrm{K}}$ by composition through $\llbracket \mu X.A \rrbracket^\sharp$.

Example: 3*-limited binary trees* Consider the recursive equation

$$\mathsf{Tree} = \mathsf{unit} + \mathsf{int} * \mathsf{Tree} * \mathsf{Tree}$$

which represents binary trees of integers. Examples include the empty leaf $(\imath_1 \star)$, and the balanced tree with 5, -3 and -6 as elements $(\imath_2(5, \imath_2(-3, \imath_1\star, \imath_1\star), \imath_2(-6, \imath_1\star, \imath_1\star)))$. The abstract lattice interpretation of 3-limited binary trees (with signs representing integers) is

$$(\llbracket \mathsf{Tree} \rrbracket^\sharp)^3(\Sigma) = \Sigma \times (\mathsf{Sign} \otimes (\llbracket \mathsf{Tree} \rrbracket^\sharp)^2(\Sigma) \otimes (\llbracket \mathsf{Tree} \rrbracket^\sharp)^2(\Sigma))$$

which defines an inductive abstract representation of binary trees that is three levels deep, and ends with $(\llbracket \mathsf{Tree} \rrbracket^\sharp)^0(\Sigma) = \Sigma$.

Concrete examples of elements of the abstract 3-limited binary tree domains include binary trees that have either depth of 0 or 2 and alternate between positive and negative elements:

$$\begin{array}{l}(\top, (+, \\ \qquad (\bot, (-, (\top, \bot), (\top, \bot))), \\ \qquad (\bot, (-, (\top, \bot), (\top, \bot)))))\end{array}$$

This abstraction captures our concrete balanced tree presented above.

Another example is the top element of the domain:

$$\begin{array}{l}(\top, (\top, \\ \qquad (\top, (\top, (\top, \top, \top)), (\top, (\top, \top, \top)))), \\ \qquad (\top, (\top, (\top, (\top, \top, \top)), (\top, (\top, \top, \top))))))\end{array}$$

Here all the elements of Σ and Sign are represented by their top elements, and the tree does so recursively until it reaches the limit.

5.2 Modular Typed Regular Tree Expressions

For practical program analyses, we often would like to go beyond prefixes and also capture inductive invariants. A classical way to capture these inductive invariants, is by relying on regular tree expressions [12] which capture the structure of

inductive types using grammars. We will show how we can build an extended strongly typed version of regular tree expressions (RTEs) in our framework. This will provide an idea on how to convert otherwise monolithic domains to be compositional. We further show how this domain can be directly related by concretization to the general solution for abstracting recursive types provided in Sect. 4; this intermediate relation is useful for compositionality of analyses and allows reusing soundness proofs based on the general solution.

In RTEs we capture inductive invariants by constructing a syntactic grammar over possible values. The RTE domain contains these syntactic grammars as elements, and is ordered by language inclusion.

Example: Positive-negative binary trees Reconsider the binary tree example from the previous section. We can represent the 3-limited alternating positive-negative tree abstraction as the following non-recursive grammar in the RTE domain:

$$P_0 ::= \text{Leaf} \qquad N_1 ::= \text{Node}(-, P_0, P_0) \qquad P_2 ::= \text{Node}(+, N_1, N_1)$$

We can further generalize the grammar above in the RTE domain, to capture the invariant inductively at any depth. We do this using the following recursive grammar:

$$P ::= \text{Leaf} \mid \text{Node}(+, N, N) \qquad N ::= \text{Node}(-, P, P)$$

Here, P and N are symbols that can be referenced recursively on the right-hand side, thus inductively describing the required invariants. Notice how the grammar contains syntactic references (P and N) deeply nested inside the constructors on the right-hand side. We would like these references to only be valid given the grammar context, so we do not refer to undefined symbols or symbols of the wrong type. Because of this, the domain is usually implemented in a monolithic fashion. Our goal is to make this domain more modular, while preserving strong type safety.

The representation of the positive-negative tree grammar in our strongly typed framework is as follows:

$$[P \mapsto (\mathsf{unit}+\mathsf{int}*X*X, (\top, (+, N, N))), \quad N \mapsto (\mathsf{unit}+\mathsf{int}*X*X, (\bot, (-, P, P)))]$$

In particular, we represent the grammar as an environment mapping variables to a pair where the first component is the target program type B (excluding function types $A \Rightarrow B$) and the second component represents the interpretation of that program type $[\![B]\!]^\sharp_{\mathrm{E}} : \mathsf{Type} \to \mathsf{Set}$ into our modular abstract domains. Formally, given a countable set of symbols $a, b \in \mathsf{Sym}$ we define an RTE pre-environment

$$\Gamma^{-\sharp} : \mathsf{RTEnv}^{-\sharp} = \mathsf{Sym} \to \{(B, t^\sharp) \mid B : \mathsf{Type}, t^\sharp \in [\![B]\!]^\sharp_{\mathrm{E}}(\mu X.B)\}$$

where the first argument to the abstract interpretation is the type of the recursive references. We define the interpretation of types as follows:

$$[\![X]\!]^\sharp_{\mathrm{E}}(B) = \{(a, B) \mid a \in \mathsf{Sym}\} \qquad [\![\mathsf{int}]\!]^\sharp_{\mathrm{E}}(B) = \mathsf{Sign}$$

$$[\![\mathsf{unit}]\!]^\sharp_{\mathrm{E}}(B) = \Sigma \qquad [\![\mu X.B_1]\!]^\sharp_{\mathrm{E}}(B) = \{(b, \mu X.B_1) \mid b \in \mathsf{Sym}\}$$

$$[\![B_1+B_2]\!]^\sharp_{\mathrm{E}}(B) = [\![B_1]\!]^\sharp_{\mathrm{E}}(B) \times [\![B_2]\!]^\sharp_{\mathrm{E}}(B) \qquad [\![B_1*B_2]\!]^\sharp_{\mathrm{E}}(B) = [\![B_1]\!]^\sharp_{\mathrm{E}}(B) \otimes [\![B_2]\!]^\sharp_{\mathrm{E}}(B)$$

Our interpretation is similar to the general interpretation given in Sect. 4, except that type variables and recursive types are replaced by typed grammar symbols. Pre-environments are not well-formed since they allow arbitrary symbols, which might have the wrong type or not be defined. We need to add a well-formedness constraint $\mathsf{WFEnv}(\Gamma^{-\sharp})$ on pre-environments $\Gamma^{-\sharp}$ to get proper environments:

$$\mathsf{WFEnv}(\Gamma^{-\sharp}) = \forall a \in \mathsf{dom}\ \Gamma^{-\sharp}.\mathsf{WFType}(\Gamma^{-\sharp}(a), \Gamma^{-\sharp})$$

Here, $\mathsf{WFType}(B, t^\sharp, \Gamma^{-\sharp})$ is a predicate[6] on type B, its abstract interpretation $t^\sharp \in [\![B]\!]^\sharp_{\mathrm{E}}(\mu X.B)$ and pre-environment $\Gamma^{-\sharp}$, that checks whether the mapped values in the pre-environment map to existing symbols of the correct type:

$$\mathsf{WFType}(\mathsf{int}, t^\sharp, \Gamma^{-\sharp}) \quad \mathsf{WFType}(\mathsf{unit}, \star, \Gamma^{-\sharp})$$

$$\mathsf{WFType}(X, (a, B), \Gamma^{-\sharp}) \iff \exists t^\sharp.\Gamma^{-\sharp}(a) = (B, t^\sharp)$$

$$\mathsf{WFType}(B_1 + B_2, (t_1^\sharp, t_2^\sharp), \Gamma^{-\sharp}) \iff \mathsf{WFType}(B_1, t_1^\sharp, \Gamma^{-\sharp}) \wedge \mathsf{WFType}(B_2, t_2^\sharp, \Gamma^{-\sharp})$$

$$\mathsf{WFType}(B_1 * B_2, (t_1^\sharp, t_2^\sharp), \Gamma^{-\sharp}) \iff \mathsf{WFType}(B_1, t_1^\sharp, \Gamma^{-\sharp}) \wedge \mathsf{WFType}(B_2, t_2^\sharp, \Gamma^{-\sharp})$$

$$\mathsf{WFType}(\mu X.B, (b, \mu X.B), \Gamma^{-\sharp}) \iff B \neq X \wedge \exists t^\sharp.\Gamma^{-\sharp}(b) = (\mu X.B, t^\sharp)$$

Abstract elements of type int and unit are always well-formed, abstract sums $B_1 + B_2$ and product $B_1 * B_2$ elements must check well-formedness recursively, and type variables X and recursive types $\mu X.B$ must check that the referenced symbols map to values of the correct type in the pre-environment. We disallow direct recursion $\mu X.X$, since they would make our concretization non-productive.

We can now define environments as pre-environments which are well-formed:

$$\Gamma^\sharp \in \mathsf{RTEnv}^\sharp = \{\Gamma^{-\sharp} \mid \Gamma^{-\sharp} \in \mathsf{RTEnv}^{-\sharp} \wedge \mathsf{WFEnv}(\Gamma^{-\sharp})\}$$

Similarly, we can define our top-level interpretation of types as abstract elements closed under an environment:

$$[\![A]\!]^\sharp_{\mathrm{RTE}} = \{(\Gamma^\sharp, t^\sharp) \mid \Gamma^\sharp \in \mathsf{RTEnv}^\sharp \wedge t^\sharp \in [\![A]\!]^\sharp_{\mathrm{E}} \wedge \mathsf{WFType}(A, t^\sharp, \Gamma^\sharp)\}$$

Our top-level types A are closed, but have a similar interpretation to open types B without the Type argument needed for recursion.

Semantic operations Most semantic operations stay the same as in the previous section, the only exception is that we must define $\mathsf{unfold}^\sharp$ and $\mathsf{fold}^\sharp$ for our RTE interpretation of recursive types. We define $\mathsf{unfold}^\sharp : [\![\mu X.B]\!]^\sharp_{\mathrm{RTE}} \to [\![B\,[\mu X.B/X]]\!]^\sharp_{\mathrm{RTE}}$ as follows:

$$\mathsf{unfold}^\sharp(\Gamma^\sharp, a) = (\Gamma^\sharp, t^\sharp) \quad \mathsf{where} \quad (B, t^\sharp) = \Gamma^\sharp(a)$$

The definition is intuitively simple: we look up the target symbol in the environment to expose the underlying structure of the recursive type.

[6] We have flattened the tuple in the first argument, to improve presentation.

Formally, we need to check a few conditions to ensure it is correct. The well-formedness condition for the environment ensures that the target symbol a is in the environment and that its mapped value $\Gamma^\sharp a$ is well-formed as well. To ensure that the result $(B, t^\sharp)$ is in the target domain $[\![B\,[\mu X.B/X]]\!]^\sharp_{\mathrm{RTE}}$, we only need to check $t^\sharp$ since $\Gamma^\sharp$ stays the same. Recall that $t^\sharp$ must be in $[\![B]\!]^\sharp_{\mathrm{E}}(\mu X.B)$ by definition of pre-environments, which means that all direct type variable references X are required to be in the set $\{(a, \mu X.B) \mid a \in \mathsf{Sym}\}$. The second component of the result domain $[\![B\,[\mu X.B/X]]\!]^\sharp_{\mathrm{RTE}}$ is in $[\![B\,[\mu X.B/X]]\!]^\sharp_{\mathrm{E}}$, which had all its direct type variable references X syntactically replaced with $\mu X.B$, and whose interpretation is exactly the same set we got from the lookup $\{(a, \mu X.B) \mid a \in \mathsf{Sym}\}$. Our interpretation of $\mathsf{unfold}^\sharp$ is therefore correct.

We now define $\mathsf{fold}^\sharp : [\![B\,[\mu X.B/X]]\!]^\sharp_{\mathrm{RTE}} \to [\![\mu X.B]\!]^\sharp_{\mathrm{RTE}}$ as follows:

$$\mathsf{fold}^\sharp(\Gamma^\sharp, t^\sharp) = (\Gamma^\sharp[a \mapsto (B, t^\sharp)], a) \quad \mathsf{where} \quad a\ \mathsf{fresh}$$

Essentially, we create a new definition in our environment pointed to by a fresh symbol a that maps to the given input structure $t^\sharp$. Since a is fresh, the well-formedness of other values in the environment is unchanged and the well-formedness of the result is immediate from the extension. It is also immediate that our result lies in the required set $\{(a, \mu X.B) \mid a \in \mathsf{Sym}\}$. Our equations from Theorem 3 are satisfied provided we quotient by equivalent grammars, since folding/unfolding grammars does not lose information.

Lattice operations The RTE forms a bounded but not complete lattice, since some sets of RTEs have non-regular trees as least upper bound. This means that there can be programs which do not have the best available abstract interpretation in this domain (their least fixed-point does not exist), and so we must settle for some over-approximating fixed-point instead.

The bottom element is $(\Gamma^\sharp, \bot)$ and the top element is $(\Gamma^\sharp, \top)$ for any environment $\Gamma^\sharp$. To allow defining the other operations [3], it is necessary to redefine the parameterized composite lattices (like products) to pass down information to their parameters. This makes definitions slightly less modular, but more general. We define $\sqsubseteq_e$ on RTEs as passing down a map $e : \mathsf{Sym} \to \mathscr{P}\mathsf{Sym}$ that dynamically maps a symbol to the set of symbols that over-approximates it in the current context. Then we can define inclusion on symbols as follows:

$$(\Gamma^\sharp, (B, a)) \sqsubseteq_e (\Gamma^\sharp, (B, b)) \iff \begin{cases} b \in e(a) \\ b \notin e(a) \wedge (\Gamma^\sharp, \Gamma^\sharp(a)) \sqsubseteq_{e[a \mapsto e(a) \cup \{b\}]} (\Gamma^\sharp, \Gamma^\sharp(b)) \end{cases}$$

The first case states that a is included in b if it is assumed so in e. In the second case the inclusion is delegated to the relevant abstract lattice as before, with the minor addition that $e[a \mapsto e(a) \cup \{b\}]$ should be passed down recursively to when new symbols are met. This shows inclusion by bisimulation, since if the right-hand side of the symbol a in $\Gamma^\sharp$ is included in the right-hand side of the symbol b, then we can safely assume that b covers at least the same cases as a.

Similarly, we pass down a partial symmetric map $u : \mathsf{Sym} \times \mathsf{Sym} \rightharpoonup \mathsf{Sym}$ to the least upper bound operation to keep track of already merged symbols:

$$(\Gamma^\sharp, (A, a)) \sqcup_u (\Gamma^\sharp, (A, b)) = (\Gamma^\sharp, c) \quad \text{if } u(a, b) = c$$

$$(\Gamma^\sharp, (A, a)) \sqcup_u (\Gamma^\sharp, (A, b)) = (\Gamma^{\sharp\prime}[c \mapsto t^\sharp], t^\sharp) \quad \text{if } \begin{cases} (a, b) \notin \mathsf{dom}\ u \\ c \text{ fresh} \\ u' = u[(a, b) \mapsto c] \\ (\Gamma^{\sharp\prime}, t^\sharp) = \\ \qquad (\Gamma^\sharp, \Gamma^\sharp(a)) \sqcup_{u'} (\Gamma^\sharp, \Gamma^\sharp(b)) \end{cases}$$

The greatest lower bound case is analogous. If the two environments of the input to the operations are different, we can simply merge them by renaming all the symbols in one of the environment and then extending the other with it.

Concretization We can define a concretization to sets of concrete values from RTEs $\gamma : [\![A]\!]^\sharp_{\mathrm{RTE}} \to \mathscr{P}[\![A]\!]$ by composition of a concretization to the intermediate abstract domain presented in Sect. 4 ($\gamma'_A : [\![A]\!]^\sharp_{\mathrm{RTE}} \to [\![A]\!]^\sharp$) and its concretization to sets of concrete types ($\gamma'' : [\![A]\!]^\sharp \to \mathscr{P}[\![A]\!]$). It is easier to define γ' than γ directly, and it allows changing abstraction of a type without requiring to redo the complete soundness proof that was required in Sect. 4.

We define $\gamma'_A : [\![A]\!]^\sharp_{\mathrm{RTE}} \to [\![A]\!]^\sharp$ co-inductively as follows:

$$\gamma'_X(\Gamma^\sharp, (a, A)) = \gamma'_A(\Gamma^\sharp, t^\sharp) \quad \text{where} \quad (A, t^\sharp) = \Gamma^\sharp(a)$$

$$\gamma'_{\mathsf{int}}(\Gamma^\sharp, n^\sharp) = n^\sharp \qquad \gamma'_{\mathsf{unit}}(\Gamma^\sharp, x^\sharp) = x^\sharp$$

$$\gamma'_{\mu X.A}(\Gamma^\sharp, (a, \mu X.B)) = \gamma'_A(\Gamma^\sharp, t^\sharp) \quad \text{where} \quad (B, t^\sharp) = \Gamma^\sharp(a)$$

$$\gamma'_{A_1+A_2}(\Gamma^\sharp, (t_1^\sharp, t_2^\sharp)) = (\gamma'_{A_1}(\Gamma^\sharp, t_1^\sharp), \gamma'_{A_2}(\Gamma^\sharp, t_2^\sharp))$$

$$\gamma'_{A_1 * A_2}(\Gamma^\sharp, (t_1^\sharp, t_2^\sharp)) = (\gamma'_{A_1}(\Gamma^\sharp, t_1^\sharp), \gamma'_{A_2}(\Gamma^\sharp, t_2^\sharp))$$

The concretization simply recursively unfolds the definitions for symbols, with interpretation of most types being direct. The definition is productive since we disallow direct recursive definitions $\mu X.X$ in the WFType predicate.

6 Implementation

We have implemented a first-order version of the expression language with its concrete and abstract semantics for k-limited trees. The implementation is in Agda [21], and shows how one could implement the ideas modularly in practice [7].

We first define the syntax of the types in Agda. The opn and cls, stratify the syntax to match the open (B) and closed (A) types from Sect. 1.

```
data State : Set where opn cls : State
```

[7] https://github.com/ahmadsalim/agda-moddom

```
data Type' : State -> Set where
  Int Bool : Type' cls
  Const : Type' cls -> Type' opn
  Var : Type' opn
  _*t_ _+t_ : forall {s} -> Type' s -> Type' s -> Type' s
  Rec : Type' opn -> Type' cls

Type : Set
Type = Type' cls

Ctx : Nat -> Set
Ctx n = Vec Type n
```

The core of our implementation depends on a modular interface of semantic operations (SemanticOps), which our language interpretation[8] relies on. It relies on a field [[_]]t to provide an interpretation of our syntax of types into concrete and abstract domains[9]. The rest of the semantic operations are specified in a strongly typed fashion with regards to that implementation, which ensures that we can modularly exchange the domains for a particular type t. We parameterize over m to allow possible general monadic computations in the computation rule for sums, E-case, since it contains function arguments. Adding new types is easy as well: we update the interface with the semantic operations for that type and implement them in the corresponding concrete and abstract interpretations; this can be done in an isolated manner—without affecting other operations—because of the strong typing of the interface.

```
record SemanticOps (m : Set -> Set) : Set1 where
  field
    [[_]]t : Type -> Set

    P-pair : forall {t s} -> [[ t ]]t -> [[ s ]]t -> [[ t *t s ]]t
    P-fst : forall {t s} -> [[ t *t s ]]t -> [[ t ]]t
    P-snd : forall {t s} -> [[ t *t s ]]t -> [[ s ]]t
    E-left : forall {t s} -> [[ t ]]t -> [[ t +t s ]]t
    E-right : forall {t s} -> [[ s ]]t -> [[ t +t s ]]t
    E-case : forall {t s w} -> [[ t +t s ]]t -> ([[ t ]]t -> m [[ w ]]t)
                                             -> ([[ s ]]t -> m [[ w ]]t) -> m [[ w ]]t
    S-abs : forall {t} -> [[ t < Rec t > ]]t -> [[ Rec t ]]t
    S-rep : forall {t} -> [[ Rec t ]]t -> [[ t < Rec t > ]]t
```

We need to use a dependently typed language like Agda to get a strongly typed interpretation, because recursive types require substitution which is a type-level operation (_<_>). An inherent challenge that arises is that it is sometimes necessary to make explicit proofs to allow type checking to succeed.

[8] In Language.Rec and Language.Semantics modules.
[9] In Domains.Concrete and Domains.Abstract modules respectively.

For example, we had to show the fundamental lemma of substitution for both the concrete and abstract interpretation of types, in order for Agda to accept our definitions of S-abs and S-rep:

```
SynSemSub : forall t t' -> Equiv [[ t < t' > ]]t ([[ t ]]t' [[ t' ]]t)
```

This proof obligation is a reasonable price to pay for achieving modularity, and often implies less work than implementing ad-hoc, monolithic domains.

7 Related Work

Cousot and Cousot [13] present a framework for compositional analysis of programs that relies on symbolic relational analysis for sharing information. Existing work on specifying inductive properties for relational domains [7, 8] requires fixed shape for inductive structures. Extending our modular construction to support relational information is future work. Rival et al. [22, 25] discuss how to provide a way to modularize symbolic memories used by pointer-manipulating by decompositing them into distinct sub-memories which share information. This is suitable if different parts of the program need to be analyzed with different abstractions. Our work instead focuses on combining domains together in a modular fashion, that allows easily changing parts of the abstractions.

Benton [5] presents a systematic mathematically sound way of deriving abstractions for strictness properties for various algebraic data types (sums, products, inductive types). Our approach is more general since it allows modular construction of a large set of abstractions by combining those for specific types.

Jensen [15] presents a generic framework to derive abstract domains for various analyses using a program logic. The framework is able to reason precisely about disjunctive program properties, also inside inductive values, but does not systematically define a Galois connection between concrete and abstract domains like we do here. Combining these techniques could be interesting future work.

Darais et al. [14] present ways to reuse program implementations for various modes of concrete and abstract interpretation. Their work focuses on using monad transformers as a mean to provide interpretation of semantic operations that can be given both concrete and abstract semantics. Similarly, Keidel et al. [17] present a compositional semantics for concrete and abstract interpretation based on category theory, including a general technique for proving soundness and implementation based on Haskell's *arrows* library. These ideas can be used well with our modular domain construction, which allows further modularity at the type level and easily replacing core operations for complex structures. Both works contribute toward systematically writing abstract interpreters: Keidel et al.'s work focuses on making it easier to prove correctness of concrete abstract interpreters, while our work focuses on providing a general mathematical framework that works modularly with inductive types and higher-order functions.

8 Conclusion

We have shown that the theory for solving recursive domain equations can be applied to abstract interpretation for systematically constructing abstract domains for recursive data types, as well as the accompanying Galois connection with the concrete domains. This extends and completes the existing theory of compositionally combining abstract domains for base types, products, sums and functions. The abstract domain provided by the solution can be further abstracted to yield computable abstractions. We demonstrate this by providing a Galois connection for k-limited trees and regular tree expressions. The framework does not capture all abstract domains that have been proposed for specific data types. In particular, relational properties between sub-structures of recursive types (*e.g.* "the elements of a list are sorted") are not captured by our compositional construction. Nevertheless, we see a great potential for the use of the framework with language specification and engineering tools for building domain-specific languages in particular.

References

1. Abbott, M.G., Altenkirch, T., Ghani, N.: Containers: Constructing strictly positive types. Theor. Comput. Sci. **342**(1), 3–27 (2005)
2. Abramsky, S., Jung, A.: Domain theory. In: Abramsky, S., Gabbay, D.M., Maibaum, T.S.E. (eds.) Handbook of Logic in Computer Science, vol. 3, pp. 1–168. Clarendon Press (1994)
3. Aiken, A., Murphy, B.R.: Implementing regular tree expressions. In: vol. 523, S.L. (ed.) FPCA 1991. pp. 427–447 (1991)
4. Awodey, S.: Category Theory. Oxford University Press, Oxford (2011)
5. Benton, P.N.: Strictness properties of lazy algebraic datatypes. In: Cousot, P., Falaschi, M., Filé, G., Rauzy, A. (eds.) WSA'93. pp. 206–217. Springer (1993)
6. Bodin, M., Gardner, P., Jensen, T., Schmitt, A.: Skeletal semantics and their interpretations. PACMPL **3**(POPL), 44:1–44:31 (2019)
7. Chang, B.E., Rival, X.: Relational inductive shape analysis. In: Necula, G.C., Wadler, P. (eds.) POPL 2008. pp. 247–260. ACM (2008)
8. Chang, B.E., Rival, X., Necula, G.C.: Shape analysis with structural invariant checkers. In: Nielson, H.R., Filé, G. (eds.) SAS 2007. pp. 384–401 (2007)
9. Cousot, P.: The calculational design of a generic abstract interpreter. In: Broy, M., Steinbrüggen, R. (eds.) Calculational System Design (1999)
10. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Graham, R.M., Harrison, M.A., Sethi, R. (eds.) POPL 1977. pp. 238–252. ACM (1977)
11. Cousot, P., Cousot, R.: Invited talk: Higher order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection, and PER analysis. In: Bal, H.E. (ed.) Proceedings of the IEEE Computer Society 1994 International Conference on Computer Languages, May 16-19, 1994, Toulouse, France. pp. 95–112. IEEE Computer Society (1994)
12. Cousot, P., Cousot, R.: Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In: FPCA 1995. pp. 170–181. ACM (1995)

13. Cousot, P., Cousot, R.: Modular static program analysis. In: Horspool, R.N. (ed.) CC 2002. pp. 159–178 (2002)
14. Darais, D., Labich, N., Nguyen, P.C., Horn, D.V.: Abstracting definitional interpreters (functional pearl). PACMPL **1**(ICFP), 12:1–12:25 (2017)
15. Jensen, T.P.: Disjunctive program analysis for algebraic data types. ACM Trans. Program. Lang. Syst. **19**(5), 751–803 (1997)
16. Journault, M., Miné, A., Ouadjaout, A.: Modular static analysis of string manipulations in C programs. In: Podelski, A. (ed.) SAS 2018. pp. 243–262 (2018)
17. Keidel, S., Poulsen, C.B., Erdweg, S.: Compositional soundness proofs of abstract interpreters. PACMPL **2**(ICFP), 72:1–72:26 (2018)
18. Midtgaard, J., Jensen, T.: A calculational approach to control-flow analysis by abstract interpretation. In: SAS. pp. 347–362 (2008)
19. Nielson, F., Nielson, H.R.: The tensor product in wadler's analysis of lists. Sci. Comput. Program. **22**(3), 327–354 (1994)
20. Nielson, F., Nielson, H.R., Hankin, C.: Principles of program analysis. Springer (1999)
21. Norell, U.: Dependently typed programming in agda. In: Advanced Functional Programming, 6th International School, AFP 2008. pp. 230–266 (2008)
22. Rival, X., Toubhans, A., Chang, B.E.: Construction of abstract domains for heterogeneous properties (position paper). In: Margaria, T., Steffen, B. (eds.) ISoLA 2014. vol. 8803, pp. 489–492. Springer (2014)
23. Streicher, T.: Domain-theoretic foundations of functional programming. World Scientific Publishing Company (2006)
24. Tarski, A.: A lattice-theoretical fixpoint theorem and its applications. Pacific J. Math. **5**(2), 285–309 (1955)
25. Toubhans, A., Chang, B.E., Rival, X.: Reduced product combination of abstract domains for shapes. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) VMCAI 2013. pp. 375–395. Springer (2013)

A Proofs

Theorem 1. *If $\rho_1 \subseteq \rho_2$ then $[\![e]\!]\rho_1 \subseteq [\![e]\!]\rho_2$*

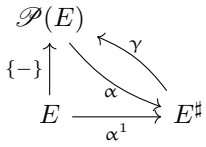
Proof. Most cases follow straightforwardly by induction over the syntax of terms e and functoriality of the powerset lifting $\mathscr{P}$. We will consider the interesting cases below:

- Case $e = x$: follows by monotonicity of premise.
- Case $e = \mathsf{case}\ e_0\ \mathsf{of}\ \mathsf{inl}\ x \to e_1 \parallel \mathsf{inr}\ y \to e_2$: follows by monotonicity of extension and induction hypothesis.
- Case $e = x \Rightarrow e_0$: follows by monotonicity of extension and induction hypothesis.
- Case $e = \mathsf{fix}\ x.e_0$: follows by monotonicity of extension, induction hypothesis and the fact that the intersection of a set of monotone functions is itself a set of monotone functions.

Proposition 1. *Let E be a set and $E^\sharp$ a complete lattice. There is a bijective correspondence between maps $\alpha^1 : E \to E^\sharp$ and Galois connections $\mathscr{P}(E) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} E^\sharp$. The correspondence maps a Galois connection to α^1 defined as $\alpha^1(x) = \alpha(\{x\})$ and a map α^1 to the Galois connection:*

$$\alpha(X \subseteq E) = \bigsqcup_{x \in X} \alpha^1(x) \qquad \gamma(e^\sharp) = \left\{ x \in E \mid \alpha^1(x) \sqsubseteq e^\sharp \right\}$$

The correspondence can be summarized in the following diagram.



Proof.

$$\begin{aligned} \alpha(X) \sqsubseteq e^\sharp &\iff \left(\bigsqcup_{x \in X} \alpha^1(x) \right) \sqsubseteq e^\sharp \\ &\iff \forall x \in X. \alpha^1(x) \sqsubseteq e^\sharp \\ &\iff X \subseteq \gamma(e^\sharp) \end{aligned}$$

Theorem 3. *Suppose the abstract interpretation of recursive types*

$$\mathsf{fold}^\sharp : [\![B\,[\mu X.B/X]]\!]^\sharp \to [\![\mu X.B]\!]^\sharp \qquad \mathsf{unfold}^\sharp : [\![\mu X.B]\!]^\sharp \to [\![B\,[\mu X.B/X]]\!]^\sharp$$

satisfies the equations

$$\mathsf{fold}^\sharp \circ \mathsf{unfold}^\sharp = \mathsf{id} \qquad \mathsf{unfold}^\sharp \circ \mathsf{fold}^\sharp \sqsupseteq \mathsf{id}$$

If $\vdash_{\mathsf{ex}} t : A$ then $[\![t]\!] \subseteq \gamma([\![t]\!]^\sharp)$, or equivalently $\alpha([\![t]\!]) \sqsubseteq [\![t]\!]^\sharp$

Theorem 3 is proved by induction on typing judgements, and must therefore be extended to open terms. This is done in the following lemma, which uses an extension of the maps γ and α to contexts defined in the obvious (pointwise) way.

Lemma 1. *Suppose $\Gamma \vdash_{\mathsf{ex}} e : A$, and that $\rho \in [\![\Gamma]\!]$ and $\rho^\sharp \in [\![\Gamma]\!]^\sharp$, are such that $\rho \subseteq \gamma(\rho^\sharp)$. If the conditions on the abstract interpretation of recursive types of Theorem 3 are satisfied, then $[\![t]\!]\rho \subseteq \gamma([\![t]\!]^\sharp\rho^\sharp)$ or, equivalently, $\alpha([\![t]\!]\rho) \subseteq [\![t]\!]^\sharp\rho^\sharp$*

Theorem 3 obviously follows from this as a special case. Before proving Lemma 1 we need a few lemmas.

Lemma 2. *If B is an open type and A a closed one, then $[\![B]\!]([\![A]\!]) = [\![B\,[A/X]]\!]$ and $[\![B]\!]^\sharp([\![A]\!]^\sharp) = [\![B\,[A/X]]\!]^\sharp$*

Proof. Easy induction on B, which we omit.

Lemma 3. *For any open type B and closed type A the following equality holds*

$$\alpha^1_B([\![A]\!]^\sharp) \circ [\![B]\!](\alpha^1_A) = \alpha^1_{B[A/X]}$$

In diagram style, this is

$$\begin{array}{ccccc} [\![B\,[A/X]]\!] & \xrightarrow{\mathsf{id}} & [\![B]\!]([\![A]\!]) & \xrightarrow{[\![B]\!](\alpha^1_A)} & [\![B]\!]([\![A]\!]^\sharp) \\ \downarrow{\scriptstyle \alpha^1_{B[A/X]}} & & & & \downarrow{\scriptstyle \alpha^1_B([\![A]\!]^\sharp)} \\ [\![B\,[A/X]]\!]^\sharp & & \xrightarrow{\qquad\qquad\mathsf{id}\qquad\qquad} & & [\![B]\!]^\sharp([\![A]\!]^\sharp) \end{array}$$

Proof. Induction on B. The cases of integers and unit type are trivial. In the case of a type variable X, we get

$$\alpha^1_X([\![A]\!]^\sharp) \circ [\![X]\!](\alpha^1_A) = \mathsf{id}_{[\![A]\!]^\sharp} \circ \alpha^1_A = \alpha^1_A = \alpha^1_{X[A/X]}$$

In the case of a product type we get

$$\begin{aligned} \alpha^1_{B_1 * B_2}([\![A]\!]^\sharp) \circ [\![B_1 * B_2]\!](\alpha^1_A)(x,y) &= \alpha^1_{B_1 * B_2}([\![A]\!]^\sharp)([\![B_1]\!](\alpha^1_A)(x), [\![B_2]\!](\alpha^1_A)(y)) \\ &= (\alpha^1_{B_1}([\![A]\!]^\sharp)([\![B_1]\!](\alpha^1_A)(x)), \alpha^1_{B_2}([\![A]\!]^\sharp)([\![B_2]\!](\alpha^1_A)(y))) \\ &= (\alpha^1_{B_1[A/X]}(x), \alpha^1_{B_2[A/X]}(y)) \\ &= \alpha^1_{B_1[A/X] * B_2[A/X]}(x,y) \\ &= \alpha^1_{(B_1 * B_2)[A/X]}(x,y) \end{aligned}$$

using the induction hypothesis in the third equality. The case for sum types is somewhat similar, except here we must branch over the input being an injection on the left or right. In the first case we get

$$\begin{aligned} \alpha^1_{B_1 + B_2}([\![A]\!]^\sharp) \circ [\![B_1 + B_2]\!](\alpha^1_A)(\imath_1(x)) &= \alpha^1_{B_1 + B_2}([\![A]\!]^\sharp) \circ (\imath_1([\![B_1]\!](\alpha^1_A)(x))) \\ &= (\alpha^1_{B_1}([\![A]\!]^\sharp)([\![B_1]\!](\alpha^1_A)(x)), \bot) \\ &= (\alpha^1_{B_1[A/X]}(x), \bot) \\ &= \alpha^1_{B_1[A/X] + B_2[A/X]}(\imath_1(x)) \\ &= \alpha^1_{(B_1 + B_2)[A/X]}(\imath_1(x)) \end{aligned}$$

again using the induction hypothesis in the third equality. The case of the input being an injection on the right is similar.

The final case is that of functions: $B = (A_1 \Rightarrow B_1)$. If $f : [\![A_1]\!] \to [\![B_1]\!]([\![A]\!]^\sharp)$ then

$$\begin{aligned}\alpha^1_{A_1 \Rightarrow B_1}([\![A]\!]^\sharp) \circ [\![A_1 \Rightarrow B_1]\!](\alpha^1_A)(f) &= \alpha^1_{A_1 \Rightarrow B_1}([\![A]\!]^\sharp)([\![B_1]\!](\alpha^1_A) \circ f) \\ &= \alpha_{B_1}([\![A]\!]^\sharp) \circ \mathscr{P}([\![B_1]\!](\alpha^1_A) \circ f) \circ \gamma_{A_1} \\ &= \alpha_{B_1}([\![A]\!]^\sharp) \circ \mathscr{P}([\![B_1]\!](\alpha^1_A)) \circ \mathscr{P}(f) \circ \gamma_{A_1}\end{aligned}$$

By the induction hypothesis

$$\alpha^1_{B_1}([\![A]\!]^\sharp) \circ [\![B_1]\!](\alpha^1_A) = \alpha^1_{B_1[A/X]}$$

which implies

$$\alpha_{B_1}([\![A]\!]^\sharp) \circ \mathscr{P}([\![B_1]\!](\alpha^1_A)) = \alpha_{B_1[A/X]}$$

since either side of this equation is the unique extension of the corresponding sides of the equation above to continuous maps. Thus, we get

$$\begin{aligned}\alpha^1_{A_1 \Rightarrow B_1}([\![A]\!]^\sharp) \circ [\![A_1 \Rightarrow B_1]\!](\alpha^1_A)(f) &= \alpha_{B_1[A/X]} \circ \mathscr{P}(f) \circ \gamma_{A_1} \\ &= \alpha^1_{A_1 \Rightarrow (B_1[A/X])}(f) \\ &= \alpha^1_{(A_1 \Rightarrow B_1)[A/X]}(f)\end{aligned}$$

proving the case and concluding the proof.

Lemma 4. *If B is an open type, then the following diagram commutes*

$$\begin{array}{ccc} \mathscr{P}([\![B\,[\mu X.B/X]]\!]) & \xrightarrow{\alpha_{B[\mu X.B/X]}} & [\![B\,[\mu X.B/X]]\!]^\sharp \\ \downarrow{\scriptstyle \mathscr{P}(\mathsf{fold})} & & \downarrow{\scriptstyle \mathsf{fold}^\sharp} \\ \mathscr{P}([\![\mu X.B]\!]) & \xrightarrow{\alpha_{\mu X.B}} & [\![\mu X.B]\!]^\sharp \end{array}$$

Proof. It suffices to show that $\alpha^1_{\mu X.B} \circ \mathsf{fold} = \mathsf{fold}^\sharp \circ \alpha^1_{B[\mu X.B/X]}$ since the two compositions in the diagram are the unique extensions of the two sides of this equation to continuous maps. By definition of $\alpha^1_{\mu X.B}$ we get

$$\alpha^1_{\mu X.B} \circ \mathsf{fold} = \mathsf{fold}^\sharp \circ \alpha^1_B([\![\mu X.B]\!]^\sharp) \circ [\![B]\!](\alpha^1_{\mu X.B})$$

By Lemma 3 the right hand side of this is equal to $\mathsf{fold}^\sharp \circ \alpha^1_{B[\mu X.B/X]}$ which concludes the proof.

We can now prove Lemma 1.

Proof (Lemma 1). By induction on typing derivation:

- Case $t = x$ trivial

– Case $t = \mathsf{fix}\ \lambda x.s.$:
By IH we get:

$$\begin{aligned}[\![s]\!]\rho[x \mapsto \gamma([\![t]\!]^\sharp\rho^\sharp)] &\subseteq \gamma([\![s]\!]^\sharp\rho^\sharp[x \mapsto [\![t]\!]^\sharp\rho^\sharp]) \\ &= \gamma([\![t]\!]^\sharp\rho^\sharp)\end{aligned}$$

So $\gamma([\![t]\!]^\sharp\rho^\sharp)$ is a post-fixpoint of $\lambda a.[\![s]\!]\rho[x \mapsto a]$ and so greater than the greatest lower bound of these ($[\![t]\!]\rho = \bigcap\{a \mid [\![s]\!]\rho[x \mapsto a] \subseteq a\}$).

– Case $t = s\ u$:
Recall that $\alpha_{A\to B} : \mathscr{P}([\![A]\!] \to [\![B]\!]) \to [\![A]\!]^\sharp \to [\![B]\!]^\sharp$ is defined as:

$$\alpha(F) = \bigsqcup_{f\in F} \alpha_B \circ \mathscr{P}(f) \circ \gamma_A$$

Then

$$\begin{aligned}\alpha_B([\![s\ u]\!]\rho) &= \alpha_B(\{f(a) \mid f \in [\![s]\!]\rho, a \in [\![u]\!]\rho\}) \\ &= \alpha_B(\bigcup_{f\in[\![s]\!]\rho} \mathscr{P}(f)([\![u]\!]\rho)) \\ &= \bigsqcup_{f\in[\![s]\!]\rho} \alpha_B(\mathscr{P}(f)([\![u]\!]\rho)) \\ &\sqsubseteq \bigsqcup_{f\in[\![s]\!]\rho} \alpha_B(\mathscr{P}(f)(\gamma_A(\alpha_A([\![u]\!]\rho)))) \\ &= \alpha_{A\to B}([\![s]\!]\rho)(\alpha([\![u]\!]\rho)) \\ &\sqsubseteq [\![s]\!]^\sharp\rho^\sharp([\![u]\!]^\sharp\rho^\sharp) \text{ By IH}\end{aligned}$$

– Case $t = \lambda x.u : A \to B$
We must show that for all $f \in [\![\lambda x.u]\!]\rho$ it holds that

$$\alpha \circ \mathscr{P}(f) \circ \gamma \sqsubseteq [\![\lambda x.u]\!]^\sharp\rho^\sharp$$

or equivalently

$$\mathscr{P}(f) \circ \gamma \sqsubseteq \gamma \circ [\![\lambda x.u]\!]^\sharp\rho^\sharp \tag{1}$$

We know by IH that for all $a^\sharp$ it holds

$$\begin{aligned}[\![u]\!]\rho[x \mapsto \gamma(a^\sharp)] &\subseteq \gamma([\![u]\!]^\sharp\rho^\sharp[x \mapsto \alpha(\gamma(a^\sharp))]) &(2)\\ &\subseteq \gamma([\![u]\!]^\sharp\rho^\sharp[x \mapsto a^\sharp]) &(3)\end{aligned}$$

To show Eq. (1) we must show that for all $a \in \gamma(a^\sharp)$ it holds:

$$\begin{aligned}f(a) &\in \gamma((\![\lambda x.u]\!]^\sharp\rho^\sharp)a^\sharp) \\ &= \gamma([\![u]\!]^\sharp\rho^\sharp[x \mapsto a^\sharp])\end{aligned}$$

The assumption $f \in [\![\lambda x.u]\!]\rho$ means precisely that $f(a) \in [\![u]\!]\rho[x \mapsto \{a\}] \subseteq [\![u]\!]\rho[x \mapsto \gamma(a^\sharp)]$ (by monotonicity), so by Eq. (3) we conclude.

– Case $t = \mathsf{fold}\ u : \mu X.A$,
The case is proved as follows

$$\begin{aligned}\alpha(\llbracket \mathsf{fold}\ u \rrbracket \rho) &= \alpha(\mathscr{P}(\mathsf{fold})(\llbracket u \rrbracket \rho)) \\ &= \mathsf{fold}^\sharp(\alpha(\llbracket u \rrbracket \rho)) \\ &\sqsubseteq \mathsf{fold}^\sharp(\llbracket u \rrbracket^\sharp \rho^\sharp) \\ &= \llbracket \mathsf{fold}\ u \rrbracket^\sharp \rho^\sharp\end{aligned}$$

using Lemma 4 for the second equality.
– Case $t = \mathsf{unfold}\ u : A\,[\mu X.A/X]$
First note that

$$\alpha \circ \mathscr{P}(\mathsf{unfold}) \sqsubseteq \mathsf{unfold}^\sharp \circ \alpha$$

since

$$\begin{aligned}\alpha \circ \mathscr{P}(\mathsf{unfold}) &\sqsubseteq \mathsf{unfold}^\sharp \circ \mathsf{fold}^\sharp \circ \alpha \circ \mathscr{P}(\mathsf{unfold}) \\ &= \mathsf{unfold}^\sharp \circ \alpha \circ \mathscr{P}(\mathsf{fold} \circ \mathsf{unfold}) \\ &= \mathsf{unfold}^\sharp \circ \alpha\end{aligned}$$

and so

$$\begin{aligned}\alpha(\llbracket \mathsf{unfold}\ u \rrbracket \rho) &= \alpha(\mathscr{P}(\mathsf{unfold})(\llbracket u \rrbracket \rho)) \\ &\sqsubseteq \mathsf{unfold}^\sharp(\alpha(\llbracket u \rrbracket \rho)) \\ &\sqsubseteq \mathsf{unfold}^\sharp(\llbracket u \rrbracket^\sharp \rho^\sharp) \text{ by IH} \\ &= \llbracket \mathsf{unfold}\ u \rrbracket^\sharp \rho^\sharp\end{aligned}$$