Submitted to MFPS 2023

A Categorical Framework for Program Semantics and Semantic Abstraction

Shin-ya Katsumata[a,1]

[a] *National Institute of Informatics*

Xavier Rival[b,2]

[b] *INRIA Paris, Département d'Informatique de l'Ecole Normale Supérieure de Paris, Université PSL, CNRS*

Jérémy Dubut[c,3]

[c] *National Institute of Advanced Industrial Science and Technology*

Abstract

Categorical semantics of type theories are often characterized as structure-preserving functors. This is because in category theory both the syntax and the domain of interpretation are uniformly treated as structured categories, so that we can express interpretations as structure-preserving functors between them. This mathematical characterization of semantics makes it convenient to manipulate and to reason about relationships between interpretations. Motivated by this success of functorial semantics, we address the question of finding a functorial analogue in *abstract interpretation*, a general framework for comparing semantics, so that we can bring similar benefits of functorial semantics to semantic abstractions used in abstract interpretation. Major differences concern the notion of interpretation that is being considered. Indeed, conventional semantics are value-based whereas abstract interpretation typically deals with more complex properties. In this paper, we propose a functorial approach to abstract interpretation and study associated fundamental concepts therein. In our approach, interpretations are expressed as oplax functors in the category of posets, and abstraction relations between interpretations are expressed as lax natural transformations representing concretizations. We present examples of these formal concepts from monadic semantics of programming languages and discuss soundness.

Keywords: semantics, abstract interpretation, oplax functors, monads.

1 Introduction

In a categorical setting, programs semantics can often be characterized as functors. In this setup, programs are viewed as morphisms and morphism composition describes how programs can be composed. As an example, in the case of typed functional programs, one may let objects be types and morphisms be functions. More precisely, a morphism from object a to object b denotes a function of type $a \to b$. Then, the semantics maps programs to morphisms between objects that interpret input and output elements into a well-chosen semantic domain. This construction is very general and accepts a wide range of semantic domains.

[1] Email: s-katsumata@nii.ac.jp
[2] Email: rival@di.ens.fr
[3] Email: jeremy.dubut@aist.go.jp

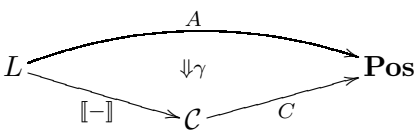
This functorial presentation of semantics is prominent in the categorical semantics of type theories and algebraic theories. There, both type theories and semantic categories are treated as categories possessing a common structure, and the semantics itself is presented as a structure-preserving functor. A typical example is the categorical semantics of the simply typed lambda-calculus, commonly studied under the well-known Curry-Howard-Lambek correspondence [22]; the calculus modulo $\beta\eta$-equality is presented as a Cartesian closed category, and its semantics in a Cartesian closed category is presented as a functor preserving finite products and exponentials. Another example is the categorical presentation of algebraic theories as a particular kind of categories with finite products (called *Lawvere theories* [23,1]), and their models as finite-product preserving functors. These categorical and syntax-free presentations of the calculus and its semantics brought significant convenience and advances to the study of type theories and their semantics. Additionally, monads turned out to be the tool of choice in order to construct semantics for effectful programs [25].

Abstract interpretation [9] provides a framework to compare program semantics of varying levels of expressiveness, and to derive sound approximations of program semantics, based on a given abstraction relation. It has been used to describe relationships across program semantics [8], program analysis [9,5,16], program transformations [13], and more. However, it is usually formalized in order theory since this presentation suffices in many applications. Therefore, it is not immediately compatible with the aforementioned categorical presentation.

Although the notion of Galois connection, which is abundantly used in abstract interpretation works [9,12], is adjunctions between posets, few works have studied a more complete description of abstract interpretation frameworks in a categorical setup. Among the works that relied on categorical tools in order to describe some specific semantic abstraction concepts for specific purposes, we can cite, Steffen et al. [29] who integrate both concrete and abstract semantics in a categorical settings in order to examine questions related to soundness and completeness, with respect to a given set of behaviors. Venet [30] uses mathematical tools that stem from category theory in order to construct specific families of abstract domains. More precisely, he applies the Grothendieck construction to generalize constructions such as reduced product and cardinal power [10]. More recently, Sergey et al. [28] took advantage of the monadic structure of a semantics of lambda-calculus to derive a static control flow analysis for a small functional language as well as an implementation in Haskell.

In this work, we seek for more comprehensive foundations for classical abstract interpretation techniques into the categorical semantics settings. We start with an interpretation of programs as morphisms in a syntactic category and semantics as functors from programs to the category of posets. We formalize and generalize the notion of collecting semantics typically used in program analysis as a decomposition of such a functor. In this setup, we integrate the notion of abstraction using some form of natural transformations between these functors. More precisely, the approximation inherent in sound, incomplete abstractions are accounted for using lax natural transformations. We show that this construction also enables the abstract interpretation of a basic language.

To achieve these goals, we build upon a categorical interpretation of programs and their semantics. In our categorical formalism, the design of an abstract semantics with respect to a denotational semantics $[\![-]\!] : L \to \mathcal{C}$ proceeds as follows. First, we turn the denotational semantics into a *functorial collecting semantics* by composing $[\![-]\!]$ with a functor $C : \mathcal{C} \to \mathbf{Pos}$ (where **Pos** denotes the category of posets and monotone functions between them), which we call *property functor*. This functor plays the role of attaching a notion of property and a direct image operation to the category $\mathcal{C}$. This step is crucial for the design of the analysis, as it fixes the concrete semantics the analysis is built upon. Then, an analysis using abstract domains over the collecting semantics $C \circ [\![-]\!]$ is expressed as an *oplax* functor $A : L \to \mathbf{Pos}$ equipped with a *lax* natural transformation $\gamma : A \to C \circ [\![-]\!]$ representing a *concretization of interpretation*:



(1)

Here, the functor A being oplax means that it only satisfies weakened functor axioms. The lax natural transformations are also weakening of natural transformations, replacing the naturality axiom to an inequality. The use of oplax functors for modeling analysis was initiated by Steffen, Jay and Mendler [29].

We adopt the same approach, and further bring some basic concepts that are not covered in [29] into the oplax functor formalism.

The common approach relies on fixing such an abstraction relation (here described by γ) and seeking for a sound, possibly approximate A that can be implemented efficiently. A natural and important question is how such an (A, γ) pair can be constructed. This can be done by extending the collecting semantics with a family of *Galois connections*, which are abundantly used in abstract interpretation, or with a family of concretization functions when best abstraction cannot be ensured.

This story naturally extends to the monadic semantics of various effectful programming languages. Indeed, as discussed earlier, the semantics of such programs is often derived using Kleisli categories of monads. Assuming a base category $\mathcal{C}$ for values and a monad T for effects, effectful programs are interpreted in the Kleisli category $\mathcal{C}_T$, and the semantics takes the form of a functor $F : L \to \mathcal{C}_T$. We then derive a *collecting semantics* by composing it with a functor $\mathcal{C}_T \to \mathbf{Pos}$, and its abstraction is given, following the lax natural transformation discussed previously. Therefore, another benefit of our approach is to simplify the design of static analyses for effectful programs, thanks to a better integration of program semantics and abstraction.

To summarize, upon the work by Steffen, Jay and Mendler [29], we formalize abstract interpretation in a functorial semantics framework. The new ingredients from [29] are the following:

(i) We show that interpretations (oplax functors) are closed under the *induction operation* by Galois connections (Theorem 4.2). This induction also comes with *concretizations of interpretations*, formulated as lax natural transformations.

(ii) We give a categorical formulation of *collecting semantics* (Section 3), which is the starting point of the development of abstract interpretations. In our formulation, a collecting semantics is an extension of a standard denotational semantics with a *property functor*, which attaches forward predicate transformers to the model category of the denotational semantics.

(iii) We present two examples of developments of abstract interpretations, one for a while language over generic computational effects (Section 3 and Section 4), and the other for the simply typed lambda calculus in Section 5. An additional result that follows from this approach is a strongest postcondition predicate transformer semantics for the while language over general monads and truth value complete lattices (Theorem 3.13). This semantics is a generalization of the strongest postcondition semantics introduced in [32].

2 Interpretation as an Oplax Functor

In this section, we set up basic definitions that serve as foundations for our integrated framework in the next sections. At this point, our main goal is to formalize the construction of semantics from programs.

2.1 Programs

Before considering semantic interpretations, we need to fix syntactic definitions. In this paper, we make the choice to also integrate programs as categorical entities, so that we can define interpretations as functors. This presentation is natural since programs can be viewed as transformations from inputs to inputs, that can also be composed like morphisms in a category. To do that, we follow a classical idea that lets a category $\mathcal{L}$ stand for the syntactic definition of programs and their inputs/outputs; more precisely,

- objects describe elements consumed/produced by programs;
- programs stand for morphisms.

As an example, in a functional setup, we may let objects be types (thus describing values) and morphisms be functions mapping values to values.

Definition 2.1 [Programs as morphisms] In the following, we express a programming language by a category L. Its objects represent types/contexts of programs, and morphisms stand for programs.

This approach has been often used to describe functional programs (see e.g. [22,14]). However, it also applies to other families of programs, such as imperative languages, that may not seem, at first, ideally adapted to a categorical approach. We illustrate this in the following examples.

Example 2.2 [A small imperative programming language] We fix a set of values $\mathbb{V}$ and a finite set of variables X. We define the category L_{w} as follows. First, we let expressions be defined by the grammar $e \coloneqq v$ (where $v \in \mathbb{V}$) $\mid x$ (where $x \in X$) $\mid e \oplus e$ (where $\oplus \in \{+, -, *, \leq, =, \ldots\}$). Second, we let programs be defined by the grammar $P \coloneqq \mathbf{skip} \mid P; P \mid x := e \mid \mathbf{if}\, e\, \{P\} \mathbf{else} \{P\} \mid \mathbf{while}\, e\, \{P\}$. Intuitively, an expression e is a value, the reading of a variable, or a binary expression, and a program is either the **skip** program that does nothing, or a sequence, or an assignment, or a condition, or a loop. The sequence construction acts like a composition. To satisfy the properties of morphism composition, we need to equate $\mathbf{skip}; P$ and $P; \mathbf{skip}$ with P (identity) and also to let sequential composition act as an associative operation (which boils down to not parenthesizing sequences). Therefore L_{w} simply follows the typical single object category describing a monoid.

As the above example is rather contrived in the sense that it omits any form of scoping, we consider a second, more sophisticated version.

Example 2.3 [Imperative programs with scoped variables] In this example, we extend Example 2.2 with a notion of scope, based on explicit variable creation and destruction operations. We keep the definition of expressions unchanged and extend that of programs with two additional constructions $\mathbf{addvar}\, x$ which adds a new local variable x and stores a default value (that we assume to be 0) into it and $\mathbf{delvar}\, x$ which ends the scope of a local variable x. Note that conventional block-based scoping can be encoded using these two operations.

We revise the construction of L_{w} in Example 2.2 and let objects be finite sets of variables. Intuitively, the object X denotes set of memories storing $|X|$-many values. To make this explicit, we stratify the set of programs by two sets of variables, that respectively denote the variables in their input and output states. More precisely, we specify the set $L_{\mathrm{wv}}(X, X')$ of programs from X to X'. Then, well-formed programs are defined inductively as follows. For any set X, $\mathbf{skip} \in L_{\mathrm{wv}}(X, X)$. Given programs $P \in L_{\mathrm{wv}}(X, X')$ and $P' \in L_{\mathrm{wv}}(X', X'')$ the sequence program $P; P'$ is in the set $L_{\mathrm{wv}}(X, X'')$. Assignment $x := e$ belongs to $L_{\mathrm{wv}}(X, X)$ for any X such that x and all variables in e are elements of X. Given two programs $P, P' \in L_{\mathrm{wv}}(X, X')$, the condition program $\mathbf{if}\, e\, \{P\} \mathbf{else} \{P'\}$ is in $L_{\mathrm{wv}}(X, X')$ when all variables in e are in X. Similarly, loop $(\mathbf{while}\, e\, \{P_{X,X}\})_{X,X}$ is defined when all variables in e are elements of X (note that the body and the loop program are morphisms from X to itself). Finally, when $x \notin X$, $(\mathbf{addvar}\, x) \in L_{\mathrm{wv}}(X, X \uplus \{x\})$ and $(\mathbf{delvar}\, x) \in L_{\mathrm{wv}}(X \uplus \{x\}, X)$. We note that the sequence case acts as composition of two morphisms respectively from X to X' and from X' to X'' and that it produces a morphism from X to X''. The resulting category is denoted by L_{wv}.

2.2 Semantic Interpretation

Based on the category of programs setup in Section 2.1, we now turn to semantic interpretations. Intuitively, a semantic interpretation should map programs into mathematical objects that describe their behavior in a more abstract manner than just syntax. For instance, the interpretation of a program P may boil down to a function that maps program input states to corresponding output states. Such an interpretation is expected to meet some compositionality property. Therefore, functors appear as the right categorical tool to define semantic interpretations. Before we spell out any definition, we consider a couple of examples.

Example 2.4 [Denotational Semantics] In this first example, we consider the language of Example 2.2. Programs are naturally regarded as actions on memories, namely functions from variables to values. We thus first define $\mathbb{M} \coloneqq [X \to \mathbb{V}]$ to be the set of memories. Since any memory state is a valid input/output for any program, the construction of Definition 2.1 requires a single object that stands for $\mathbb{M}$. This action will be given as the semantics in the next Example. We observe that not all programs terminate, which entails that one input state may not correspond to exactly one output state. Therefore, following the classical approach to denotational semantics, we interpret the unique object of L into the set $\mathbb{M}_{\perp\!\!\!\perp} \coloneqq \mathbb{M} \uplus \{\perp\!\!\!\perp\}$ where $\perp\!\!\!\perp$ stands for non-termination. Then, the interpretation of a program P is the $\perp\!\!\!\perp$-strict function $[\![P]\!]$ from $\mathbb{M}_{\perp\!\!\!\perp}$ into itself. We remark that $\mathbb{M}_{\perp\!\!\!\perp}$ is often interpreted as a poset, where $\forall m \in \mathbb{M}, \perp\!\!\!\perp \sqsubseteq m$ and that $\perp\!\!\!\perp$-strict functions are monotone functions in that set. Such interpretations compose well in the sense that the interpretation of $P_0; P_1$ coincides with the composition of the interpretation of P_1 with that of P_0. Therefore we may convert it into a functor $[\![-]\!] : L_{\mathrm{w}} \to \mathbf{Pos}$. This is a *denotational semantics* of L_{w}, interpreting programs as *actions on states* (including $\perp\!\!\!\perp$). This type of semantics will be studied in

Section 3.1.

This functorial semantics can be extended to the imperative language with variable scoping operators L_{wv} in Example 2.3. In this language, by putting $\mathbb{M}_X := [X \to \mathbb{V}]$, program should be viewed a strict continuous function from $(\mathbb{M}_X)_\perp$ into $(\mathbb{M}_{X'})_\perp$ for two given finite sets of variables X, X'. Note that, in this settings, (**addvar** x) is well defined since it initializes the new variable x with default value 0, as specified in Example 2.3.

Example 2.5 [Collecting semantics] Another type of semantics is to interpret programs as actions on *properties* on states. There are various ways to represent them, and one way is by sets of states. Then the semantics of programs are expressed as functions from sets of states into sets of states. Such a semantics is usually called *collecting semantics* and is one of the fundamental semantics in abstract interpretation. Specifically, we interpret the unique object of L_{w} with the poset $(2^{\mathbb{M}}, \subseteq)$ and a program P with a $\cup$-preserving function $[\![P]\!]_c : (2^{\mathbb{M}}, \subseteq) \to (2^{\mathbb{M}}, \subseteq)$. Typically, $[\![P]\!]_c$ is defined by induction over the syntax of programs and is compositional in the same sense as $[\![P]\!]$ (Example 2.4). It thus determines a functor $[\![-]\!]_c : L_w \to \mathbf{Pos}$. It can also be derived rather systematically from $[\![P]\!]$ - see Section 3.3.

Example 2.6 [Interval analysis] While the semantics shown in Example 2.5 is suitable as a starting point to study static analyzers, it is not computable. Thus, we propose to consider a second interpretation, using abstract properties rather than sets of states [9]. We consider here the interval abstraction of [9], assuming values are machine integers. In this setup a set of states is described either with $\perp$ (which denotes the empty set) or with a function from variables into intervals with integer bounds (that could be the minimum or maximum representable machine integers). Therefore, an interpretation of a program P may be defined as an abstract semantics $(\!|P|\!)$ that maps an abstract pre-condition into a sound abstract post-condition: given any abstract state $m^\sharp$ and any memory m that satisfies the constraints in $m^\sharp$, all states in $[\![P]\!](\{m\})$ are described by $(\!|P|\!)(m^\sharp)$.

Nevertheless, this construction may not be compositional in the strict sense of denotational semantics. Indeed, the analysis of a composite program may be (and often is) more precise than the composition of the analyses of the sub-programs, in the following sense: we say that an analysis $(\!|P_0|\!)$ is more precise than an analysis $(\!|P_1|\!)$ when $(\!|P_0|\!)$ computes properties that are logically stronger than those computed than $(\!|P_1|\!)$. Indeed, let us consider P_0 be $x := 4 * x - 2$ and P_1 be **if** $x \leq 0$ $\{x = -x\}$**else**$\{$**skip**$\}$. Moreover, we let $m^\sharp$ map x to interval $[0, 1]$. Then, $(\!|P_0|\!)(x) = [-2, 2]$ and $(\!|P_1|\!) \circ (\!|P_0|\!)(x) = [0, 2]$. However, if we consider concrete executions starting with $x \in [0, 1]$, only value 2 may be observed as an output, thus a more precise analysis $(\!|P_0; P_1|\!)$ that maps $m^\sharp$ to $[x \mapsto [2, 2]]$ is sound.

The first remark that follows from the last two examples is that **Pos** provides a natural setup for property-based semantics. Indeed, objects of **Pos** allow to account for collections of program inputs or outputs, ordered with inclusion, with the usual meaning that smaller elements account for fewer behaviors.

The second remark is that composition may not always be exact when considering static analyses. Indeed, while the concrete semantics of Example 2.4 is compositional, Example 2.6 shows that static analyses may not be. Though, common practice ensures that most analyses satisfy a lax form of compositionality: the analysis of the composition of two programs may be implemented so that it gives somewhat more precise results than that of the composition of the analyses but is not expected to produce worse results. We comment on the limitations of this view in Remark 2.8.

To formalize these remarks, we rely on the order-enrichment on the category **Pos** of posets and monotone functions. All monotone functions $f, g : X \to Y$ in **Pos** are ordered in a pointwise manner, making **Pos** an enriched category over **Pos** itself. Intuitively, the order $f \leq g$ means that "g is *more abstract* than f"/"g is *coarser* than f". Based on this terminology, we define semantic interpretations:

Definition 2.7 [Interpretations] A *semantic interpretation* (or, for short, *interpretation*) F consists of an object mapping $F : \mathrm{Obj}(L) \to \mathrm{Obj}(\mathbf{Pos})$ and a morphism mapping $F_{X,Y} : L(X, Y) \to \mathbf{Pos}(FX, FY)$ such that the following inequalities hold:

$$F(\mathrm{id}) \leq \mathrm{id}, \quad F(f_1 \circ f_2) \leq Ff_1 \circ Ff_2.$$

We say that the interpretation is *normal* if $F(\mathrm{id}_X) = \mathrm{id}_X$ and *functorial* if it is normal and $F(f_1 \circ f_2) = Ff_1 \circ Ff_2$.

Such a weakened form of functor is known as *oplax functors*. The above definition makes sense when **Pos** is replaced with some other **Pos**-enriched category K. In fact, Steffen, Jay and Mendler considered this general definition of interpretation in [29], and their intention is to pick a suitable K for each analysis task. On the other hand, in this paper we fix the codomain of interpretation to **Pos**.

The semantics studied in Example 2.4 defines a functorial interpretation. The analysis of Example 2.6 defines an interpretation that is normal but not functorial. This latter situation was also commented by Steffen, Jay and Mendler in [29]: indeed, they also introduce an oplax interpretation and remark that strictness analysis is generally not functorial.

Remark 2.8 [Limitations] As stated above, not all semantics satisfy the property of Definition 2.7, and in particular, some static analyses fail to satisfy it. As an example, it is possible to set up a weaker form of interval analysis (Example 2.6) that gives up all precision when applied to programs with updates to more than k variables. If $k = 1$, P_0 is $x = 0$ and P_1 is $y = 1$, although these programs are trivial, $(\!|P_0; P_1|\!)$ drops all information on x, y whereas $(\!|P_1|\!) \circ (\!|P_0|\!)$ is trivially analyzed in a precise manner, thus we do not have $(\!|P_0 \circ P_1|\!) \leq (\!|P_1|\!) \circ (\!|P_0|\!)$. Obviously, this definition is very contrived, and a conventional implementation of interval analysis achieves the oplaxness property stated in Definition 2.7.

We introduce a partial order between interpretations. This order $F \leq G$ means that F, G agree on the interpretation of objects in L, but F interprets programs with better precision than G.

Definition 2.9 [Partial Order between Interpretations] Given two interpretations $F, G : L \to \mathbf{Pos}$, we write $F \leq G$ if $F(a) = G(a)$ for any $a \in L$ and $F(f) \leq G(f)$ for any $f : a \to b$ in L.

This order will be used to compare abstract interpretations derived by Galois connections and its over-approximations (Theorem 4.2,4.3,5.1).

3 Functorial Collecting Semantics

We have set-up a formulation of abstract semantics as oplax functors. Typically, the development of abstract semantics is initiated from a concrete semantics called *collecting semantics*. We therefore express collecting semantics in our categorical setting.

The spirit of the collecting semantics is to interpret the behavior of programs as actions on collections of inputs, rather than single inputs. Typically, such a collection is chosen to represent a *property* on inputs, hence below we use the word *property* to mean a collection of inputs, or a collection of elements in a set X in general. Properties on X are ordered by the inclusion, forming the poset $(2^X, \subseteq)$ (where 2^X denotes the set of subsets of X).

Let us consider a simple example of collecting semantics. We consider a set-theoretic denotational semantics of a deterministic and terminating programming language L. It is defined by a functor $[\![-]\!] : L \to \mathbf{Set}$ mapping a morphism $P : a \to b$ in L representing a program into a function $[\![P]\!] : [\![a]\!] \to [\![b]\!]$. Then, the *collecting semantics* associated to this denotational semantics is the monotone function $[\![P]\!]_c : (2^{[\![a]\!]}, \subseteq) \to (2^{[\![b]\!]}, \subseteq)$ defined by $[\![P]\!]_c(U) \coloneqq \{[\![P]\!](x) \mid x \in U\}$; the right hand side is the *direct image* of a property $U \subseteq [\![a]\!]$ by $[\![P]\!]$. We then regard the collecting semantics as a functor $[\![-]\!]_c : L \to \mathbf{Pos}$.

We categorically analyze this definition of collecting semantics as follows. We introduce the *covariant powerset functor* $Q : \mathbf{Set} \to \mathbf{Pos}$ defined by: $Q(X) \coloneqq (2^X, \subseteq)$ and $Q(f)(U) \coloneqq \{f(x) \mid x \in U\}$. The role of this functor is to assign to each $X \in \mathbf{Set}$ the poset of properties on X, and to each morphism f in **Set** the *direct image operation* associated to f. Then we notice the equality $[\![a]\!]_c = Q([\![a]\!])$ for each object $a \in L$ and $[\![P]\!]_c = Q([\![P]\!])$ for each morphism P in L, which amounts to the following functor equality:

$$[\![-]\!]_c \quad = \quad L \xrightarrow{[\![-]\!]} \mathbf{Set} \xrightarrow{Q} \mathbf{Pos}\,.$$

This functorial presentation of the collecting semantics suggests us to generalize the middle category to an arbitrary category $\mathcal{C}$ instead of **Set** so that we can define the concept of collecting semantics for general categorical semantics of programming languages. However, we need to replace the covariant powerset functor $Q : \mathbf{Set} \to \mathbf{Pos}$ with something else because it is specific to **Set**. We therefore need to supply a functor $C : \mathcal{C} \to \mathbf{Pos}$ that plays the same role as Q; it assigns to each object $X \in \mathcal{C}$ a poset CX of

properties on X, whose elements abstractly represent properties on X, and to each $f : X \to Y$ a monotone function $Cf : CX \to CY$ representing the direct image operation. Based on this observation, we derive the following definition of collecting semantics.

Definition 3.1 A *functorial collecting semantics* of a language L (regarded as a category) consists of a category $\mathcal{C}$ and two functors:

$$L \xrightarrow{[\![-]\!]} \mathcal{C} \xrightarrow{C} \mathbf{Pos}\,,$$

where $[\![-]\!]$ is called a *denotational semantics* and C is called a *property functor* (for $\mathcal{C}$). The composite $C \circ [\![-]\!] : L \to \mathbf{Pos}$ itself is also called the functorial collecting semantics (with respect to $[\![-]\!]$).

This definition of functorial collecting semantics reflects our view that concrete semantics for initiating abstract interpretation is *synthesized* from a denotational semantics $[\![-]\!] : L \to \mathcal{C}$ by composing a *property functor* $C : \mathcal{C} \to \mathbf{Pos}$. However, in some situations one may directly construct a collecting semantics without denotational semantics —in this case we simply put $C = \mathrm{Id}$.

Remark 3.2 The readers who are familiar with fibrational category theory might notice that the property functor $C : \mathcal{C} \to \mathbf{Pos}$ bijectively corresponds to a *posetal opfibration* (a functor $c : \mathcal{E} \to \mathcal{C}$ from some category $\mathcal{E}$ such that c satisfies the *opcartesian lifting property*; see e.g. [20, Section 9.1]). Therefore C may be replaced by a posetal opfibration $c : \mathcal{E} \to \mathcal{C}$. In this setting, the functorial collecting semantics interprets a morphism $f : a \to b$ in L as the *pushforward* $[\![f]\!]_* : \mathcal{E}_{[\![a]\!]} \to \mathcal{E}_{[\![b]\!]}$ between fibre posets.

3.1 Denotational Semantics

In this section, we focus on the first functor and look at several examples of denotational semantics.

Example 3.3 An extreme example of denotational semantics is the identity functor. This means that the semantics of types/contexts and programs are themselves.

A second non-trivial family of examples relates to *monadic semantics*. Good references to the definitions of *monads* and *Kleisli categories* can be found in [24] and [25]. The Kleisli category of a monad T on $\mathcal{C}$ is denoted by $\mathcal{C}_T$. We let $\bullet$ denote the composition of morphisms in Kleisli categories. Among examples of monads on **Set**, we can cite 1) the *maybe monad* that joins an extra element to a given set, and is defined by $MX = X \uplus \{*\}$, 2) the powerset monad which maps each set to its powerset and is defined by $PX = 2^X$, and 3) the monad of probability subdistributions defined by $D_s X = \{\mu : X \to [0,1] \mid \mu \text{ is a countable subdist. on } X\}$. Before we look at monadic semantics, we fix a class of monads that is required to enable a least fixpoint definition of the semantics of while commands.

Definition 3.4 A *while-monad* on **Set** consists of a monad $(T, \eta, (-)^{\#})$ on **Set** and an ω-cpo $(\sqsubseteq_X, \bot\!\!\!\bot_X)$ on each TX. The sup of an ω-chain $\{x_i\}_{i\in\mathbb{N}}$ in TX is denoted by $\bigsqcup x_i$. [4] We define $\sqsubseteq_{X,Y}$ as the pointwise order on $\mathbf{Set}_T(X,Y)$ given by $f \sqsubseteq_{X,Y} g \iff \forall x \in X\ .\ f(x) \sqsubseteq_Y g(x)$, and $\bot\!\!\!\bot_{X,Y}$ as the least function $\lambda x\ .\ \bot\!\!\!\bot_Y$. The data of the while-monad should satisfy:

(i) the Kleisli composition $\bullet$ is monotone and ω-continuous in each argument with respect to $\sqsubseteq_{X,Y}$, and
(ii) additionally, it is strict in the second argument: $f \bullet \bot\!\!\!\bot_{X,Y} = \bot\!\!\!\bot_{X,Z}$ for any $f \in \mathbf{Set}_T(Y,Z)$.

This is a simplification and specialization of the *order-enriched monad* in [18]. The order $\sqsubseteq$ is to compare the definedness of elements/functions in the sense of domain theory. We later see another order for truth values, and these two orders are independent in general. We do not require $\bullet$ to be strict in the first argument. This is because we may sometimes want to distinguish the divergence after some effect from pure divergence. Consider a program $\texttt{tick};\texttt{diverge}$. Then its monadic semantics will be $\bot\!\!\!\bot_{\mathbb{M},\mathbb{M}} \bullet [\![\texttt{tick}]\!]$, which we may want to distinguish from the silent divergence $\bot\!\!\!\bot_{\mathbb{M},\mathbb{M}}$.

Examples of while-monads on **Set** include the powerset monad P with the set inclusion order, the maybe monad M with the flat order making $\iota_2(*)$ the least element, and the countable subdistribution monad D_s with the pointwise mass order.

We now give a generic set-theoretic monadic semantics of the while language in Example 2.2.

[4] In this article, the least element of a poset is denoted by $\bot\!\!\!\bot_X$ if it corresponds to non-termination, while it is denoted by $\bot$ if it corresponds to the falsity (Example 3.9).

Example 3.5 [Monadic Semantics of While Language] We give a semantics of the while language in $\mathbf{Set}_T$. Let $(T, \eta, (-)^\#, \sqsubseteq, \perp\!\!\!\perp)$ be a while-monad on $\mathbf{Set}$. In this semantics, we allow assignment commands $x := e$ to perform some computational effects. We therefore let the interpretation of the command $x := e$ be the morphism $[\![x := e]\!] \in \mathbf{Set}_T(\mathbb{M}, \mathbb{M})$. We also assume an interpretation of the boolean expression b as a function $[\![b]\!] \in \mathbf{Set}(\mathbb{M}, \{\mathrm{ff}, \mathrm{tt}\})$ into the two-points set. Then, the interpretation of programs is given by

$$[\![\mathbf{skip}]\!]_T := \eta_{\mathbb{M}} \qquad [\![P; P']\!]_T := [\![P']\!]_T \bullet [\![P]\!]_T \qquad [\![x := e]\!]_T := [\![x := e]\!]$$
$$[\![\mathbf{if}\, b\, \{P_1\} \mathbf{else} \{P_2\}]\!]_T := \lambda\rho \,.\, \text{if } [\![b]\!](\rho) = \mathrm{tt} \text{ then } [\![P_1]\!]_T(\rho) \text{ else } [\![P_2]\!]_T(\rho)$$
$$[\![\mathbf{while}\, e\, \{P\}]\!]_T := \mu\Phi \quad \text{where} \quad \Phi(f) := \lambda\rho \,.\, \text{if } [\![b]\!](\rho) = \mathrm{tt} \text{ then } f \bullet [\![P]\!]_T(\rho) \text{ else } \eta_{\Omega^{\mathbb{M}}}(\rho)$$

Here $\mu\Phi$ is the least fixpoint of Φ, and $\eta_{\Omega^{\mathbb{M}}}$ is a component of the unit natural transformation of the monad T. We regard this interpretation as a functor $[\![-]\!]_T : L_{\mathrm{w}} \to \mathbf{Set}_T$. When T is the maybe monad M, $[\![-]\!]_M$ interprets programs as partial functions, while when T is the powerset monad P, $[\![-]\!]_P$ may be regarded as interpreting programs as binary relations.

Example 3.6 To interpret the while language L_{wv} with variable addition and deletion (Example 2.3), we may adjust the semantics in the previous example as follows. First, we let $[\![-]\!]_T$ map each object $X \in L_{\mathrm{wv}}$, which is a finite set of variables, to the set $[\![X]\!]_T := \mathbb{M}_X$ of memories over X (Example 2.4). It is then almost straightforward to modify $[\![-]\!]_T$ and let it map a program $P \in L_{\mathrm{wv}}(X, X')$ to a morphism of type $\mathbb{M}_X \to \mathbb{M}_{X'}$ in $\mathbf{Set}_T$. We interpret two additional commands **addvar** and **delvar** by

$$[\![(\mathbf{addvar}\ x)_{X, X \uplus \{x\}}]\!]_T(\rho) = \eta_{[\![X]\!] \uplus \{x\}}(\rho\{x \mapsto 0\}) \qquad [\![(\mathbf{delvar}\ x)_{X \uplus \{x\}, X}]\!]_T(\rho) = \eta_{[\![X]\!]}(\rho - x),$$

where 0 is the presupposed default value for new variables (Example 2.3) and $\rho - x$ is the environment obtained by removing x from the domain of ρ. Overall, we obtain a revised functor $[\![-]\!]_T : L_{\mathrm{wv}} \to \mathbf{Set}_T$.

3.2 Property Functor

We next see some property functors so that we can form functorial collecting semantics of denotational semantics in the previous section.

Example 3.7 An extreme example of a property functor is the identity functor on $\mathbf{Pos}$. This assigns to each poset X the poset of properties consisting of "being less than or equal to x" for each $x \in X$.

Example 3.8 The category $\mathbf{Rel}$ of sets and binary relations between them is the host category for various relational semantics of programs. For instance, the relational semantics of the while language interprets a program as an endorelation on $\mathbb{M}$. Recall that each binary relation $f \in \mathbf{Rel}(X, Y)$ determines a $\cup$-preserving function $f_S(U) := \{j \mid i \in U \wedge (i, j) \in f\} : (2^X, \subseteq) \to (2^Y, \subseteq)$. We turn this into a functor $I^P : \mathbf{Rel} \to \mathbf{CLat}_\vee$, where $\mathbf{CLat}_\vee$ is the subcategory of $\mathbf{Pos}$ consisting of complete lattices and join-preserving functions between them, by $I^P(X) := (2^X, \subseteq)$ and $I^P(f) := f_S$. Clearly we have the subcategory inclusion $\iota : \mathbf{CLat}_\vee \to \mathbf{Pos}$, so overall we obtain a property functor for $\mathbf{Rel}$ as the composite $\iota \circ I^P : \mathbf{Rel} \to \mathbf{CLat}_\vee \to \mathbf{Pos}$.

Example 3.9 Recall that $\mathbf{Rel}$ is isomorphic to the Kleisli category $\mathbf{Set}_P$ of the powerset monad P. Thus the property functor in the previous example may be seen as the composite $\iota \circ I^P : \mathbf{Set}_P \to \mathbf{CLat}_\vee \to \mathbf{Pos}$. In this example, we generalize this diagram in two directions: one is to replace P with a $\mathbf{Set}$-monad $(T, \eta, (-)^\#)$, and the other is to adopt complete-lattice valued predicates as properties. We then derive a property functor using the *strongest postcondition predicate transformer* for Kleisli categories studied in [2].

We first take a complete lattice $(\Omega, \leq)$ for truth values. We use the symbols $\bot, \top, \vee, \wedge$ to mean the least/greatest elements and joins/meets of Ω. We then regard a function $\phi : X \to \Omega$ as an *Ω-valued property on X*. Such properties form a complete lattice $\Omega^X := (\mathbf{Set}(X, \Omega), \leq_X)$, where $\leq_X$ is the pointwise order. The lattice operations on Ω^X is written by $\wedge_X, \vee_X$, etc. We note that the mapping $X \mapsto \Omega^X$ extends to a functor of type $\mathbf{Set}^{op} \to \mathbf{CLat}$ (the category of complete lattices and complete lattice homomorphisms).

We next take a *T-algebra* (or *Eilenberg-Moore T-algebra*) $o : T\Omega \to \Omega$ (see [24, Section VI.2]), and assume that o is *meet-preserving* in the following sense: the function $\lambda\phi \,.\, o \circ T(\phi)$ belongs to $\mathbf{CLat}_\wedge(\Omega^X, \Omega^{T(X)})$,

the homset of the category $\mathbf{CLat}_\wedge$ of complete lattices and meet-preserving functions. Such a T-algebra induces the *weakest precondition predicate transformer* $wp^o(f) \in \mathbf{CLat}_\wedge(\Omega^Y, \Omega^X)$ for each $f \in \mathbf{Set}_T(X, Y)$. It is given by $wp^o(f) \coloneqq \lambda\phi \,.\, o \circ T\phi \circ f$ [2, Corollary 4.6]. The mapping $f \mapsto wp^o(f)$ extends to a functor of type $\mathbf{Set}_T^{op} \to \mathbf{CLat}_\wedge$, thanks to the Eilenberg-Moore axioms.

We then take the left adjoint $sp^o(f) \in \mathbf{CLat}_\vee(\Omega^X, \Omega^Y)$ of $wp^o(f)$, which we call the *strongest postcondition predicate transformer* [2, Example 4.11]. We extend the mapping $f \mapsto sp^o(f)$ to a functor $I^o : \mathbf{Set}_T \to \mathbf{CLat}_\vee$ given by $I^o(X) \coloneqq \Omega^X$ and $I^o(f) \coloneqq sp^o(f)$. In this way we obtain a property functor $\iota \circ I^o : \mathbf{Set}_T \to \mathbf{CLat}_\vee \to \mathbf{Pos}$.

We see a few examples of meet-preserving T-algebras where $\Omega = \{\bot \leq \top\}$. Now Ω^X is the poset of characteristic functions, and we identify it with the poset $(2^X, \subseteq)$ of subsets of X.

(i) For the powerset monad P, there is only one T-algebra $o : P\Omega \to \Omega$ preserving meets: $o(U) = \top \iff \bot \notin U$ [2, Example 5.3]. The derived property functor is isomorphic to the one in Example 3.8.

(ii) For the maybe monad M, there is only one T-algebra $o : M\Omega \to \Omega$ preserving meets: the one sending $\iota_2(\perp\!\!\!\perp)$ to $\top$. The strongest postcondition predicate transformer (hence property functor I^o) satisfies, for $f \in \mathbf{Set}_M(X, Y)$, $sp^o(f) = \lambda\phi \,.\, \{y \in Y \mid \exists x \in X \,.\, x \in \phi \wedge f(x) = \eta_Y(y)\}$.

Example 3.10 Any category $\mathcal{C}$ with a functor $U : \mathcal{C} \to \mathbf{Set}$ has a property functor given by the composite $Q \circ U : \mathcal{C} \to \mathbf{Set} \to \mathbf{Pos}$ with the covariant powerset functor Q.

For example, let $(T, \eta, (-)^\#)$ be a monad on $\mathbf{Set}$. Its Kleisli category $\mathbf{Set}_T$ comes with the right adjoint functor $K^T : \mathbf{Set}_T \to \mathbf{Set}$ given by $K^T(X) \coloneqq T(X)$ and $K^T(f) \coloneqq f^\#$ [24, Theorem VI.1]. We then compose this with the covariant powerset functor Q and obtain a property functor $Q \circ K^T : \mathbf{Set}_T \to \mathbf{Set} \to \mathbf{Pos}$.

By letting T be the powerset monad P, we obtain the property functor that assigns to a set X the set of *hyperproperties* over X in the sense of Clarkson and Schneider [6]. The object part of $Q \circ K^T$ sends a set X to the poset $(2^{2^X}, \subseteq)$ of hyperproperties on X.

3.3 Putting Together

We have seen several denotational semantics and property functors. By combining them we obtain functorial collecting semantics.

We consider a functorial collecting semantics of the while language L_{w} with 1) the monadic denotational semantics $[\![-]\!]_T : L_{\mathrm{w}} \to \mathbf{Set}_T$ in Example 3.5, and 2) the property functor $\iota \circ I^o : \mathbf{Set}_T \to \mathbf{Pos}$ given by the strongest postcondition predicate transformer sp^o in Example 3.9. The functorial collecting semantics interprets a program P as $[\![P]\!]_c \coloneqq \iota \circ I^o([\![P]\!]_T) = sp^o([\![P]\!]_T)$.

We address the question of whether this collecting semantics can be given inductively. Let $(T, \eta, (-)^\#, \sqsubseteq, \perp\!\!\!\perp)$ be the while-monad on $\mathbf{Set}$ used in the monadic denotational semantics, $(\Omega, \leq)$ be the complete lattice, and $o : T\Omega \to \Omega$ be the meet-preserving T-algebra used in the property functor. We remark that these two orders represent two independent notions: $\sqsubseteq$ represents the definedness order of computations in the sense of domain theory, while $\leq$ represents the strength of truth values in the sense of algebraic logic. As noted in [12], we emphasize these two orders may be different in general. We also use different symbols for these two orders: joins for a definedness order is denoted by $\sqcup$.

The following theorem states that the collecting semantics of while-free programs can be inductively given.

Theorem 3.11 *The functorial collecting semantics $[\![-]\!]_c$ satisfies:*

$$[\![\mathbf{skip}]\!]_c = \mathrm{id}_{\Omega^{\mathbb{M}}} \qquad [\![P; P']\!]_c = [\![P']\!]_c \circ [\![P]\!]_c \qquad [\![x := e]\!]_c = sp^o([\![x := e]\!])$$
$$[\![\mathbf{if}\, b\, \{P_1\} \mathbf{else} \{P_2\}]\!]_c = \lambda\phi \,.\, [\![P_1]\!]_c(\phi \wedge_{\mathbb{M}} [b = \mathrm{tt}]) \vee_{\mathbb{M}} [\![P_2]\!]_c(\phi \wedge_{\mathbb{M}} [b = \mathrm{ff}]).$$

Here, $[b = v] : \mathbb{M} \to \{\bot_{\mathbb{M}}, \top_{\mathbb{M}}\} \subseteq \Omega^{\mathbb{M}}$ is the function defined by $[b = v](\rho) = \top_{\mathbb{M}}$ if and only if $[\![b]\!](\rho) = v$.

The next question is whether $[\![\mathbf{while}\, b\, \{P\}]\!]_c \in \mathbf{CLat}_\vee(\Omega^{\mathbb{M}}, \Omega^{\mathbb{M}})$ can be computed by the least fixpoint of some functional, say Ψ, definable by P and b. Since $[\![\mathbf{while}\, b\, \{P\}]\!]_T$ is the least fixpoint of a functional Φ on $\mathbf{Set}_T(\mathbb{M}, \mathbb{M})$ (see Example 3.5), it suffices to show that 1) $sp^o : \mathbf{Set}_T(\mathbb{M}, \mathbb{M}) \to \mathbf{CLat}_\vee(\Omega^{\mathbb{M}}, \Omega^{\mathbb{M}})$

is ω-continuous and strict, and 2) $\Psi \circ sp^o = sp^o \circ \Phi$. Regarding 1, it is equivalent to the continuity of $o : T\Omega \to \Omega$ in the following sense. It relates the ω-lub in the definedness order and the ω-lub in the truth value order.

Proposition 3.12 *The following hold:*

(i) $sp^o(\bigsqcup f_i) = \bigvee sp^o(f_i)$ *holds for any* $X, Y \in \mathbf{Set}_T$ *and* ω*-chain* f_i *in hom-*ω*-cpo* $(\mathbf{Set}_T(X,Y), \sqsubseteq_{X,Y})$ *if and only if* $o(\bigsqcup c_i) = \bigwedge(o(c_i))$ *holds for any* ω*-chain* c_i *in the* ω*-CPO* $(T\Omega, \sqsubseteq_\Omega)$.

(ii) $sp^o(\perp\!\!\!\perp_{X,Y}) = \lambda\phi \,.\, \bot_Y$ *if and only if* $o(\perp\!\!\!\perp_\Omega) = \top$.

Proof We here prove (i); (ii) can be proved similarly. We actually prove that the following are equivalent.

(a) $sp^o\left(\bigsqcup f_i\right) = \bigvee sp^o(f_i)$ for any $X, Y \in \mathbf{Set}_T$ and ω-chain f_i in hom-ω-CPO $(\mathbf{Set}_T(X,Y), \sqsubseteq_{X,Y})$.

(b) $wp^o\left(\bigsqcup f_i\right) = \bigwedge wp^o(f_i)$ for any $X, Y \in \mathbf{Set}_T$ and ω-chain f_i in hom-ω-CPO $(\mathbf{Set}_T(X,Y), \sqsubseteq_{X,Y})$.

(c) $o\left(\bigsqcup c_i\right) = \bigwedge(o(c_i))$ for any ω-chain c_i in the ω-CPO $(T\Omega, \sqsubseteq_\Omega)$.

(a) $\iff$ (b) Let f_i be an ω-chain in $(\mathbf{Set}_T(X,Y), \sqsubseteq_{X,Y})$. Then from $sp^o(f_i) \dashv wp^o(f_i)$, we obtain:

$$\begin{aligned} sp^o\left(\bigsqcup f_i\right) = \bigvee sp^o(f_i) &\iff \left(\forall \phi, \psi \,.\, sp^o\left(\bigsqcup f_i\right)(\phi) \leq_Y \psi \iff \bigvee sp^o(f_i)(\phi) \leq_Y \psi\right) \\ &\iff \left(\forall \phi, \psi \,.\, \phi \leq_X wp^o\left(\bigsqcup f_i\right)(\psi) \iff \phi \leq_X \bigwedge wp^o(f_i)(\psi)\right) \\ &\iff wp^o\left(\bigsqcup f_i\right) = \bigwedge wp^o(f_i). \end{aligned}$$

(b) $\implies$ (c) Let c_i be an ω-chain in $(T\Omega, \sqsubseteq_\Omega)$. Notice the isomorphism $(T\Omega, \sqsubseteq_\Omega) \cong (\mathbf{Set}_T(1,\Omega), \sqsubseteq_{1,\Omega})$. We thus identify c_i as an ω-chain in $(\mathbf{Set}_T(1,\Omega), \sqsubseteq_{1,\Omega})$. Then

$$o \circ \left(\bigsqcup c_i\right) = wp^o\left(\bigsqcup c_i\right)(\mathrm{id}_\Omega) = \bigwedge wp^o(c_i)(\mathrm{id}_\Omega) = \bigwedge o \circ c_i.$$

(c) $\implies$ (b) Let f_i be an ω-chain in $(\mathbf{Set}_T(X,Y), \sqsubseteq_{X,Y})$. Then for any $\phi : Y \to \Omega$ and $x \in X$, we have

$$\begin{aligned} &wp^o\left(\bigsqcup f_i\right)(\phi)(x) = o \circ T\phi \circ \left(\bigsqcup f_i\right)(x) = o\left(T\phi\left(\bigsqcup f_i(x)\right)\right) = o\left(\bigsqcup T\phi(f_i(x))\right) \\ &= \bigwedge o(T\phi(f_i(x))) = \bigwedge(wp^o(f_i)(\phi)(x)) = \left(\bigwedge wp^o(f_i)\right)(\phi)(x). \end{aligned}$$

Notice that $T\phi$ is continuous as T is a while monad. Therefore $wp^o\left(\bigsqcup f_i\right) = \bigwedge wp^o(f_i)$. □

We thus say that o *makes* sp^o ω*-continuous and strict* if it is a strict ω-continuous function of type $(T\Omega, \sqsubseteq_\Omega)^{op} \to (\Omega, \leq)$.

Theorem 3.13 *If* o *makes* sp^o ω*-continuous and strict, the functorial collecting semantics* $[\![-]\!]_c$ *satisfies*

$$[\![\mathbf{while}\, b\, \{P\}]\!]_c = \mu\Psi \quad \textit{where} \quad \Psi(f) = \lambda\phi \,.\, (f \circ [\![P]\!]_c(\phi \wedge_{\mathbb{M}} [b = \mathrm{tt}])) \vee_{\Omega^{\mathbb{M}}} (\phi \wedge_{\mathbb{M}} [b = \mathrm{ff}]).$$

Proof It sufficies to show $sp^o(\mu\Phi) = \mu\Psi$. It is easy to verify that Ψ is a join-preserving endofunction on the pointwise-order complete lattice $\mathbf{CLat}_\vee(\Omega^{\mathbb{M}}, \Omega^{\mathbb{M}})$. To show that $(-) \wedge_{\mathbb{M}} [b = v]$ preserves joins, we use the fact that $[b = v]$ takes only values in $\{\bot_{\mathbb{M}}, \top_{\mathbb{M}}\}$. By calculation, we have $\Psi \circ sp^o = sp^o \circ \Phi$. Since sp^o is ω-continuous and strict, we obtain $sp^o(\mu\Phi) = \mu\Psi$. □

We see some examples of meet-preserving T-algebra making sp^o ω-continuous and strict.

(i) For the powerset while-monad P on $\mathbf{Set}$ and $\Omega = \{\bot \leq \top\}$, the meet-preserving T-algebra o in Example 3.9 makes sp^o ω-continuous and strict. The functorial collecting semantics $[\![-]\!]_c$ coincides with the standard one in the literature.

(ii) For the powerset while-monad P on $\mathbf{Set}$ and $\Omega = ([0,\infty], \leq)$, the inf-operation $\inf : P[0,\infty] \to [0,\infty]$ is a meet-preserving T-algebra making sp^o ω-continuous and strict. The functorial collecting semantics

$[\![-]\!]_c$ coincides with the *quantitative strongest postcondition* in [32]. By taking the opposite complete lattice $([0,\infty],\geq)$ for Ω and replacing inf with sup, the functorial collecting semantics $[\![-]\!]_c$ coincides with the *quantitative strongest liberal postcondition* in [32].

(iii) For the maybe while-monad M and $\Omega = \{\bot \leq \top\}$, the meet-preserving T-algebra o in Example 3.9 makes sp^o ω-continuous and strict.

Example 3.14 We consider a functorial collecting semantics of the while language L_{w} with the monadic denotational semantics $[\![-]\!]_T : L_{\mathrm{w}} \to \mathbf{Set}_T$ in Example 3.5 and the property functor $Q \circ K^T : \mathbf{Set}_T \to \mathbf{Pos}$ in Example 3.10. The interpretation $Q \circ K^T([\![P]\!]_T) : 2^{T\mathbb{M}} \to 2^{T\mathbb{M}}$ of a program P by this functorial collecting semantics satisfies $Q \circ K^T([\![P]\!]_T)(U) = \{[\![P]\!]_T^{\#}(c) \mid c \in U\}$, for any $U \in 2^{T\mathbb{M}}$, where $(-)^{\#}$ is the Kleisli lifting of the monad T. When T is the powerset monad P, this functorial collecting semantics appears in Asaf et al.'s study on *hypercollecting semantics* [3, Section 4]. The definition of their hypercollecting semantics is partially inductive [5], and is an over-approximation of the above functorial collecting semantics; see the proof of Theorem 1 of their paper.

4 Semantic Abstraction

After discussing the role of collecting semantics in detail in Section 3, we now consider semantic abstraction and derivation of abstract semantics from a reference (e.g., collecting) semantics.

4.1 Abstraction Relations Between Domains and Interpretations

So far, we have studied the definition of program semantics independently from one another and have not considered comparing them quite yet. Abstract interpretation [9] is specifically motivated with semantic comparison so as to tie properties that may be proved with one to statements involving another. Therefore, we consider the comparison of program semantics here.

Several forms of *abstraction relations* have been proposed, including Galois connections [9], concretization functions without assuming the existence of a best abstraction [10], abstraction functions without assuming the existence of a concretization, or binary relations [12,11]. In categorical terms, a *Galois connection* $\alpha \dashv \gamma : A \to C$ is a pair of monotone functions $\alpha : C \to A$ and $\gamma : A \to C$ between posets such that the equivalence $\alpha(a) \leq c \iff a \leq \gamma(c)$ holds for any $a \in A, c \in C$. It is called a *Galois insertion* if $\alpha \circ \gamma = id$.

Definition 4.1 Let $A, C : L \to \mathbf{Pos}$ be interpretations.

A *concretization of domains* from A to C is a family $\{\gamma_a : A(a) \to C(a)\}_{a \in L}$ of monotone functions. We say that it is a *concretization of interpretation* from A to C if it satisfies $C(f) \circ \gamma_a \leq \gamma_b \circ A(f)$ for any $f \in L(a,b)$. Dually, an *abstraction of domains* from C to A is a family $\{\alpha_a : C(a) \to A(a)\}_{a \in L}$ of monotone functions. We say that it is an *abstraction of interpretation* from C to A if it satisfies $\alpha_b \circ C(f) \leq A(f) \circ \alpha_a$ for any $f \in L(a,b)$. A concretization (resp., abstraction) of interpretations is called *complete* if the above inequality is an equality.

Remark that this definition sets up *two* notions of concretization (and the same for abstractions): the notion of concretization of domains ties only semantic domains, just like the concretizations in [9] do whereas concretization of interpretation tie not only domains (via a concretization of domains) but also interpretations, thus providing an interpretation soundness statement as typically sought to design a static analysis.

In categorical terminology, a concretization of interpretation is exactly a *lax natural transformation* from A to C, and an abstraction of interpretation is exactly an *oplax natural transformation* from C to A. [6] Both concretizations and abstractions of interpretation express that one interpretation is more abstract than the other.

[5] The hypercollecting semantics in [3, Section 4] is not inductive at the interpretation of conditional expressions.

[6] The original definition of (op)lax natural transformation consists of families of 1-cells and 2-cells. In the current context, however, there is at most one choice of 2-cells for (op)lax natural transformations between oplax functors into **Pos**. Therefore we here treat the concept of (op)lax natural transformation as a property on families of 1-cells.

Moreover, as in [9], we remark that the same abstraction relation may be described simultaneously both by concretization of domains and by an abstraction of domains; this strong correspondence between the two then defines a Galois connection.

We now explain how we develop a sound abstract interpretation of a functorial collecting semantics $(\llbracket - \rrbracket, C)$ in our categorical framework. The ultimate goal is to construct an interpretation A together with a concretization of interpretation, as shown in the diagram (1). The development process is broken into the following steps:

(i) We give the object part of A; this corresponds to designing a domain of interpretation for each type/context $a \in L$.

(ii) We give a concretization of domains $\{\gamma_a : A(a) \to C\llbracket a \rrbracket\}$, relating the domain of interpretation of types/contexts and that of the collecting semantics.

(iii) We give the morphism part of A; this corresponds to designing how each program $P : a \to b$ transfers abstract properties in $A(a)$ to those in $A(b)$. In our categorical framework this should respect the oplax functoriality in Definition 2.7.

(iv) We check if the constructed interpretation A satisfies soundness in the following sense: for any program $P : a \to b$ in L and an abstract predicate $\phi \in A(a)$, we have $C(\llbracket P \rrbracket)(\gamma_a(\phi)) \leq \gamma_b(A(P)(\phi))$. This is equivalent to showing that γ being a concretization of interpretation from A to the functorial collecting semantics $C \circ \llbracket - \rrbracket$.

The delicate part of this process is to find a right combination of $A(a)$, γ_a and $A(P)$ to achieve the soundness, as well as the expressiveness of abstract properties and the effectiveness of the semantics (when implementing it on a computer). In many cases, it is possible to simultaneously carry out the third and fourth steps, and to derive A from the interpretation $C \circ \llbracket - \rrbracket$ and the abstraction relation, so as to achieve a sound A.

In the rest of the section we see examples of concretization of interpretation. A bigger example of the development of abstraction is given for the denotational semantics of the lambda calculus (Section 5).

4.2 The Case of Galois Connections

As discussed earlier, Galois connections are one of the fundamental constructions that are used in abstract interpretation to describe abstraction relations. In presence of such a strong connection, the semantic induction technique of an interpretation that was mentioned in the previous subsection can be made even more systematic. We now study it in our categorical framework.

Theorem 4.2 *Let $C : L \to \mathbf{Pos}$ be an (resp. normal) interpretation, $\{A(a)\}_{a \in L}$ be a family of posets indexed by objects in L, and $G \coloneqq \{\alpha_a \dashv \gamma_a : A(a) \to C(a)\}_{a \in L}$ be a family of Galois connections (resp. insertions). We define mappings on L-objects and L-morphisms by $C^G(a) \coloneqq A(a)$ and $C^G(f) \coloneqq \alpha_b \circ Cf \circ \gamma_a \quad (f : a \to b)$.*

(i) *$C^G : L \to \mathbf{Pos}$ is a (resp. normal) interpretation, which we call the interpretation* induced by *G.*

(ii) *γ is a concretization of interpretation from C^G to C. Moreover, for any interpretation $A' : L \to \mathbf{Pos}$ such that $A'(a) = A(a)$, if γ is a concretization of interpretation from A' to C, then $C^G \leq A'$.*

(iii) *α is an abstraction of interpretation from C to C^G. Moreover, for any interpretation $A' : L \to \mathbf{Pos}$ such that $A'(a) = A(a)$, if α is an abstraction of interpretation from C to A', then $C^G \leq A'$.*

Proof (i) $C^G(\mathrm{id}_a) \leq \mathrm{id}_{Ca}$ is immediate by $\alpha_a \circ \gamma_a \leq \mathrm{id}_{Ca}$. We show the other inequality:

$$C^G(g \circ f) = \alpha_c \circ Cg \circ Cf \circ \gamma_a \leq \alpha_c \circ Cg \circ \gamma_b \circ \alpha_b \circ Cf \circ \gamma_a = C^G(g) \circ C^G(f).$$

(ii) We show γ is a lax natural transformation from C^G to C, that is, $C(f) \circ \gamma_a \leq \gamma_b \circ C^G(f)$. This is evident as $C(f) \circ \gamma_a \leq \gamma_b \circ \alpha_b \circ C(f) \circ \gamma_a$. Next, let $A' : L \to \mathbf{Pos}$ be an interpretation such that $A'(a) = A(a)$ and assume that γ_a is a concretization of interpretation from A' to $\mathbf{Pos}$. This amounts to $C(f) \circ \gamma_a \leq \gamma_b \circ A'(f)$ for any $f : a \to b$ in L. Since α_b is a left adjoint of γ_b, this is equivalent to $\alpha_b \circ C(f) \circ \gamma_a \leq A'(f)$, that is, $C^G(f) \leq A'(f)$. (iii) can be proved similarly. □

When considering a program $f : a \to b$, the interpretation $\alpha_b \circ Cf \circ \gamma_a$ is often called *best abstract transformer*. The above theorem says that C^G is the most precise interpretation making γ a concretization of interpretation from C^G to C, as well as α an abstraction of interpretation from C to C^G.

The induction of abstract semantics by Galois connections is a fundamental operation in abstract interpretation, and our categorical framework of abstract interpretation accommodates it by the above theorem. Employing oplax functors as interpretations (Definition 2.7) is crucial here, because ordinary functors are not closed under the induction operation by Galois connection.

An abstraction of an interpretation in a complete lattice defining a Galois connection.

The semantics induced by a family of Galois connections would not have an inductive characterization due to its oplaxness. The common practice is to derive the best abstract transformers (or approximate ones) for the basic constructs of a language, then extend them to the whole language by induction. We discuss this approach using the functorial collecting semantics of the while language in Section 3.3 using a meet-preserving Eilenberg-Moore T-algebra $o : T\Omega \to \Omega$ and a Galois connection $G \coloneqq \alpha \dashv \gamma : A \to (\Omega^{\mathbb{M}}, \leq)$ with a general complete lattice $(A, \wedge_A, \vee_A)$; later we restrict A so that the abstract interpretation can be computed in a finite means. We apply this Galois connection to the functorial collecting semantics $[\![-]\!]_c$ in Section 3.3, and induce an abstract interpretation $[\![-]\!]_c^G : L_{\mathrm{w}} \to \mathbf{Pos}$, which is an oplax functor. It interprets a program P as a monotone function $\alpha \circ [\![P]\!]_c \circ \gamma : A \to A$, but it does not enjoy an inductive characterization, that is, $[\![P]\!]_c^G$ is not expressible by the interpretation of subprograms in P by $[\![-]\!]_c^G$. We therefore inductively construct another interpretation $(\!|-|\!)$ that uses the best abstract transformers at assignment commands:

$$(\!|\mathbf{skip}|\!) \coloneqq \mathrm{id}_A \qquad (\!|P; P'|\!) \coloneqq (\!|P'|\!) \circ (\!|P|\!) \qquad (\!|x := e|\!) \coloneqq [\![x := e]\!]_c^G (= \alpha \circ sp^o([\![x := e]\!]) \circ \gamma)$$
$$(\!|\mathbf{if}\, b\, \{P_1\} \mathbf{else} \{P_2\}|\!) \coloneqq \lambda\phi\ .\ (\!|P_1|\!)(\phi \wedge_A \alpha([b = \mathrm{tt}])) \vee_A (\!|P_2|\!)(\phi \wedge_A \alpha([b = \mathrm{ff}]))$$
$$(\!|\mathbf{while}\, b\, \{P\}|\!) \coloneqq \mu\Theta \quad \text{where} \quad \Theta(f) \coloneqq \lambda\phi\ .\ (f \circ (\!|P|\!)(\phi \wedge_A \alpha([b = \mathrm{tt}]))) \vee_A (\phi \wedge_A \alpha([b = \mathrm{ff}]))$$

Recall that $sp^o([\![x := e]\!])$ denotes the strongest postcondition predicate transformer for assignments (Example 3.9), and $[b = v]$ denotes the predicate representing the condition tests (Theorem 3.11). In the interpretation of **while**, we use the complete lattice structure on the homset $\mathbf{Pos}(A, A)$ with the pointwise order.

Theorem 4.3 *Suppose that o makes sp ω-continuous and strict (hence Theorem 3.13 holds). The interpretation $(\!|-|\!) : L_{\mathrm{w}} \to \mathbf{Pos}$ is functorial and $[\![-]\!]_c^G \leq (\!|-|\!)$ holds.*

Proof The functoriality of $(\!|-|\!)$ is obvious by definition. We show $\alpha \circ [\![P_0]\!]_c \circ \gamma \leq (\!|P_0|\!)$ by induction on the structure of P_0. We omit subscript of $\wedge, \vee$. The cases $P_0 = \mathbf{skip}, (P; P'), (x := e)$ are easy. The case $P_0 = \mathbf{if}\, b\, \{P_1\} \mathbf{else} \{P_2\}$ is shown as follows.

$$\begin{aligned}
(\!|\mathbf{if}\, b\, \{P_1\} \mathbf{else} \{P_2\}|\!) &= \lambda\phi.(\!|P_1|\!)(\phi \wedge \alpha([b = \mathrm{tt}])) \vee (\!|P_2|\!)(\phi \wedge \alpha([b = \mathrm{ff}])) \\
\{\text{IH}\} &\geq \lambda\phi.\alpha \circ [\![P_1]\!]_c \circ \gamma(\phi \wedge \alpha([b = \mathrm{tt}])) \vee \alpha \circ [\![P_2]\!]_c \circ \gamma(\phi \wedge \alpha([b = \mathrm{ff}])) \\
\{\gamma \text{ meet-pres.}\} &= \lambda\phi.\alpha \circ [\![P_1]\!]_c(\gamma(\phi) \wedge \gamma(\alpha([b = \mathrm{tt}]))) \vee \alpha \circ [\![P_2]\!]_c(\gamma(\phi) \wedge \gamma(\alpha([b = \mathrm{ff}]))) \\
\{\text{unit law and } \alpha \text{ join-pres.}\} &\geq \lambda\phi.\alpha([\![P_1]\!]_c(\gamma(\phi) \wedge [b = \mathrm{tt}]) \vee [\![P_2]\!]_c(\gamma(\phi) \wedge [b = \mathrm{ff}])) \\
\{\text{definition}\} &= \alpha \circ [\![\mathbf{if}\, b\, \{P_1\} \mathbf{else} \{P_2\}]\!]_c \circ \gamma
\end{aligned}$$

To show the case $P_0 = \mathbf{while}\, b\, \{P\}$, we compare Ψ defined in Theorem 3.13 and Θ used in the definition of $(\!|\mathbf{while}\, b\, \{P\}|\!)$. We first show $\alpha \circ \Psi(f) \circ \gamma \leq \Theta(\alpha \circ f \circ \gamma)$.

$$\begin{aligned}
\alpha \circ \Psi(f) \circ \gamma &= \lambda\phi.\alpha(f([\![P]\!]_c(\gamma(\phi) \wedge [b = \mathrm{tt}])) \vee (\gamma(\phi) \wedge [b = \mathrm{ff}])) \\
\{\text{unit law and } \alpha \text{ join-pres.}\} &\leq \lambda\phi.(\alpha \circ f \circ [\![P]\!]_c(\gamma(\phi) \wedge \gamma(\alpha([b = \mathrm{tt}])))) \vee \alpha(\gamma(\phi) \wedge \gamma(\alpha([b = \mathrm{ff}]))) \\
\{\gamma \text{ meet-pres.}\} &= \lambda\phi.(\alpha \circ f \circ [\![P]\!]_c \circ \gamma(\phi \wedge \alpha([b = \mathrm{tt}]))) \vee \alpha(\gamma(\phi \wedge \alpha([b = \mathrm{ff}]))) \\
\{\text{unit and counit law}\} &\leq \lambda\phi.(\alpha \circ f \circ \gamma \circ \alpha \circ [\![P]\!]_c \circ \gamma(\phi \wedge \alpha([b = \mathrm{tt}]))) \vee (\phi \wedge \alpha([b = \mathrm{ff}])) \\
\{\text{IH}\} &\leq \lambda\phi.(\alpha \circ f \circ \gamma \circ (\!|P|\!)(\phi \wedge \alpha([b = \mathrm{tt}]))) \vee (\phi \wedge \alpha([b = \mathrm{ff}])) \\
&= \Theta(\alpha \circ f \circ \gamma).
\end{aligned}$$

Since $\alpha \dashv \gamma$, the operation $\alpha \circ (-) \circ \gamma : (\mathbf{Pos}(A, A), \leq) \to (\mathbf{Pos}(\Omega^{\mathbb{M}}, \Omega^{\mathbb{M}}), \leq)$ is strict and continuous. Therefore

$$\begin{aligned}(\!|\mathbf{while}\, b\, \{P\}|\!) &= \bigvee \Theta^n(\bot) = \bigvee \Theta^n(\alpha \circ \bot \circ \gamma) \geq \bigvee (\alpha \circ \Psi^n(\bot) \circ \gamma) = \alpha \circ \left(\bigvee \Psi^n(\bot)\right) \circ \gamma \\ &= \alpha \circ [\![\mathbf{while}\, b\, \{P\}]\!]_c \circ \gamma.\end{aligned}$$

□

This result is generic with respect to monads, the interpretation of (effectful) assignment commands, truth values and Galois connections. This theorem gives a functorial over-approximation of the best abstract interpretation derived from Galois connections. We see a similar story with the simply typed lambda calculus in Section 5. In the following paragraphs, we discuss how Theorem 4.3 paves the way to a computable static analysis.

Evaluation of the abstraction of an interpretation with a finite height domain.
When the abstract domain is a complete lattice with the finite chain property (which asserts that any totally ordered subset of the lattice is finite), Theorem 4.3 provides an algorithm to compute the abstract semantics that it defines [9]. Indeed, the complete lattice property and the finite chain property respectively ensure the definition and the termination of the computation of $(\!|\mathbf{while}\, b\, \{P\}|\!)$. A classical analysis that falls into this case is the analysis with the lattice of constants [21].

Evaluation of the abstraction of an interpretation using fixpoint over-approximation.
In practice, few abstract domains satisfy the properties that were required in the previous paragraph. First, the complete lattice property often fails to hold. Then, the existence of least upper bounds for any family of abstract elements can then not be guaranteed. Second, even when any family of abstract elements has a least upper bound, the ω-chain generated by $(\!|\mathbf{while}\, b\, \{P\}|\!)$ may not converge after finite iterations, hence its least upper bound may not be computable. As an example, this occurs in the case of the abstract domain of intervals [9]: as it is possible to find an infinite sequence of intervals $I_0 \subset I_1 \subset \ldots I_n \subset I_{n+1} \subset \ldots$, the definition of $(\!|\mathbf{while}\, b\, \{P\}|\!)$ that was provided earlier does not terminate in general.

Such cases are typically addressed by specific fixpoint approximation abstract operators. The most common instance is *widening* [9] and consists of a binary operator $\triangledown$ that over-approximates concrete unions and ensures termination in the sense that any sequence of applications of widening stabilizes after finitely many iterates. We now detail how our framework accommodates such analyses. We use the same notations as in the previous definition of $(\!|.|\!)$ and assume a program P, a condition b, and an abstract element ϕ. Then, we define $W_n^\phi := \bigvee_A^{i=0\ldots n} \tau^i(\phi)$ where $\tau(x) := (\!|P|\!)(x \wedge_A \alpha([b = \mathrm{tt}]))$. Interpolation operators such as widening produce an over-approximation of all the elements of this sequence in terms of $\leq$. As an example, in the case of widening, the analysis computes a sequence defined by $V_0^\phi := W_0^\phi$ and $V_{n+1}^\phi := V_n^\phi \triangledown \tau(V_n^\phi)$. This sequence converges to an element that we note V^ϕ. Then the analysis produces $(\!|\mathbf{while}\, b\, \{P\}|\!)(\phi) := V^\phi \wedge_A \alpha([b = \mathrm{ff}])$.

Category of abstractions of interpretations.
We end this section by introducing the category of *abstractions* of an interpretation. This can be viewed as a categorical understanding of the so-called *lattice of abstractions*. Fix an interpretation of a category L, that is, an oplax functor $C\colon L \to \mathbf{Pos}$. This interpretation would typically be a functorial collecting semantics as in Section 3. The category of abstractions of C is the category where an object is a concretization of interpretation $\{\gamma_a : A(a) \to C(a)\}_{a \in L}$ from an interpretation A to C, and a morphism from $\{\gamma_a : A(a) \to C(a)\}_{a \in L}$ to $\{\gamma'_a : A'(a) \to C(a)\}_{a \in L}$ is a concretization of interpretation $\{\delta_a : A(a) \to A'(a)\}_{a \in L}$ such that for any object a of L, $\gamma'_a \circ \delta_a \geq \gamma_a$, saying that $\gamma'_a \circ \delta_a$ is more abstract than γ_a. We will denote it by $\mathrm{Abs}(C)$. In words, $\mathrm{Abs}(C)$ is the oplax slice category over C of the category of interpretations of L and concretizations between them.

Some standard constructions in abstract interpretation can be lifted to categorical structures of $\mathrm{Abs}(C)$, provided that C enjoys certain properties. When C factorizes through the category of meet-semilattices

and meet-preserving maps, the category $\mathrm{Abs}(C)$ has binary products. It is given by the *Cartesian product abstract domain* (see e.g. [7]). Concretely, given two concretizations $\{\gamma_a : A(a) \to C(a)\}_{a\in L}$ and $\{\gamma'_a : A'(a) \to C(a)\}_{a\in L}$, their binary product is given by the interpretation $A \times A'$ and the concretization $\{\delta_a : A(a) \times A'(a) \to C(a)\}_{a\in L}$ where $\delta_a(u, v) = \gamma_a(u) \wedge \gamma'_a(v)$. It remains to be seen what other categorical structures are available on $\mathrm{Abs}(C)$.

5 Abstracting Denotational Semantics of λ-Calculus

In this section, we develop another kind of example to demonstrate the modularity of our theory of abstractions. Examples discussed so far only consist of imperative languages with various semantics and interpretations. Here, we describe interpretations for the simply typed lambda calculus over a higher-order signature, as a larger case study to demonstrate that our theory also copes with functional languages. In this study, we adopt rather simple *non-relational abstraction* of domains, that is, the abstract domain for product types is the product of abstract domains for component types.

5.1 The Language Category: the Free Cartesian Closed Category

Given a set B, we define $Typ(B)$ to be the set of types generated from B with the type constructors for the unit type 1, the binary product type $\times$ and the arrow type $\to$. We specify the simply typed lambda calculus by a *higher-order signature*. It consists of a set B of base types, a set O of constants and a function $typ : O \to Typ(B)$ assigning a type to each constant. We do not consider equational axioms on constants. In the rest of Section 5, we let Π be a higher-order signature (B, O, typ).

We write $\lambda(\Pi)$ for the free Cartesian closed category (CCC for short) generated from Π [14]. An object of $\lambda(\Pi)$ is a type in $Typ(B)$, and a morphism from τ to τ' is a $\beta\eta$-equivalence class of a term-in-context $x : \tau \vdash M : \tau'$ in the simply typed lambda calculus over Π.

5.2 Functorial Collecting Semantics for the Lambda Calculus

We next give a functorial collecting semantics for $\lambda(\Pi)$. We first recall the standard functorial semantics of the simply typed lambda calculus in a Cartesian closed category (CCC for short). Let $(\mathcal{C}, 1_{\mathcal{C}}, \times_{\mathcal{C}}, \Rightarrow_{\mathcal{C}})$ be a CCC. A Π-*structure* in $\mathcal{C}$ consists of 1) an assignment $[\![-]\!]_0 : B \to \mathcal{C}$ of a $\mathcal{C}$-object to each base type, and 2) an assignment $[\![c]\!]_0 : 1_{\mathcal{C}} \to [\![typ(c)]\!]$ of a $\mathcal{C}$-morphism to each constant $c \in O$; here $[\![-]\!] : Typ(B) \to \mathcal{C}$ is the inductive extension of $[\![-]\!]_0 : B \to \mathcal{C}$ by

$$[\![b]\!] \coloneqq [\![b]\!]_0, \qquad [\![1]\!] \coloneqq 1_{\mathcal{C}}, \qquad [\![\tau_1 \times \tau_2]\!] \coloneqq [\![\tau_1]\!] \times_{\mathcal{C}} [\![\tau_2]\!], \qquad [\![\tau_1 \to \tau_2]\!] \coloneqq [\![\tau_1]\!] \Rightarrow_{\mathcal{C}} [\![\tau_2]\!].$$

The Π-structure induces the Cartesian closed functor[7] $[\![-]\!] : \lambda(\Pi) \to \mathcal{C}$ [14].

For our functorial collecting semantics, we employ the CCC **Set** and fix a Π-structure in **Set**, then take the induced Cartesian closed functor $[\![-]\!]$ as a denotational semantics. For the property functor, we take the covariant powerset functor $Q : \mathbf{Set} \to \mathbf{Pos}$ from Section 3. To summarize, we take the following functorial collecting semantics: $Q \circ [\![-]\!] : \lambda(\Pi) \to \mathbf{Set} \to \mathbf{Pos}$.

5.3 Abstracting Collecting Semantics by Galois Connections

Thanks to Theorem 4.2, to construct an abstract semantics of $Q([\![-]\!])$, it suffices to give a Galois connection of the form $\alpha_\tau \dashv \gamma_\tau : A(\tau) \to Q([\![\tau]\!])$ for every type $\tau \in Typ(B)$. We do so by first assuming, for each *base* type $b \in B$, 1) a poset $A_0(b)$ of abstract base type properties, and 2) a Galois connection $\alpha_b \dashv \gamma_b : A_0(b) \to Q([\![b]\!])$. For instance, when there is a base type $nat \in B$ for natural numbers and it is interpreted as the set $\mathbb{N}$ of natural numbers, we may take the poset of intervals (including the empty one) over $\mathbb{N}$ for $A_0(nat)$, and take the Galois connection for interval abstraction for $\alpha_{nat} \dashv \gamma_{nat} : A_0(nat) \to Q(\mathbb{N})$.

[7] Here it means a functor strictly preserving finite products and exponentials.

We then inductively extend the mapping $A_0 : B \to \mathbf{Pos}$ to the one $A : Typ(B) \to \mathbf{Pos}$ by the Cartesian closed structure of **Pos**:

$$A(b) \coloneqq A_0(b), \quad A(1) \coloneqq 1_{\mathbf{Pos}}, \quad A(\tau_1 \times \tau_2) \coloneqq A(\tau_1) \times_{\mathbf{Pos}} A(\tau_2), \quad A(\tau_1 \to \tau_2) \coloneqq A(\tau_1) \Rightarrow_{\mathbf{Pos}} A(\tau_2).$$

We next construct a type-indexed family of Galois connections. Below $\alpha_{\tau_i} \dashv \gamma_{\tau_i} : A(\tau_i) \to Q(\llbracket \tau_i \rrbracket)$ are Galois connections for $i = 1, 2$. (For $\alpha_1 \dashv \gamma_1 : 1_{\mathbf{Pos}} \to Q(\llbracket 1 \rrbracket)$ we take the unique one.)

$$A(\tau_1 \times \tau_2) \underset{\alpha_{\tau_1} \times \alpha_{\tau_2}}{\overset{\gamma_{\tau_1} \times \gamma_{\tau_2}}{\underset{\longleftarrow}{\overset{\longrightarrow}{\bot}}}} Q(\llbracket \tau_1 \rrbracket) \times_{\mathbf{Pos}} Q(\llbracket \tau_2 \rrbracket) \underset{\alpha^{\times}}{\overset{\gamma^{\times}}{\underset{\longleftarrow}{\overset{\longrightarrow}{\bot}}}} Q(\llbracket \tau_1 \times \tau_2 \rrbracket)$$

$$A(\tau_1 \to \tau_2) \underset{\alpha_{\tau_2} \circ \_ \circ \gamma_{\tau_1}}{\overset{\gamma_{\tau_2} \circ \_ \circ \alpha_{\tau_1}}{\underset{\longleftarrow}{\overset{\longrightarrow}{\bot}}}} Q(\llbracket \tau_1 \rrbracket) \Rightarrow_{\mathbf{Pos}} Q(\llbracket \tau_2 \rrbracket) \underset{\alpha^{\Rightarrow}}{\overset{\gamma^{\Rightarrow}}{\underset{\longleftarrow}{\overset{\longrightarrow}{\bot}}}} Q(\llbracket \tau_1 \to \tau_2 \rrbracket)$$

Here, Galois connections $\alpha^{\times} \dashv \gamma^{\times}$ and $\alpha^{\Rightarrow} \dashv \gamma^{\Rightarrow}$ are given by

$$\alpha^{\times}(U) \coloneqq (Q(\pi_1)(U), Q(\pi_2)(U)) \qquad \gamma^{\times}(U, V) \coloneqq U \times V$$
$$\alpha^{\Rightarrow}(F) \coloneqq \lambda U \,.\, \bigcup_{f \in F} Qf(U) \qquad \gamma^{\Rightarrow}(g) \coloneqq \{f \mid \forall U \subseteq \llbracket \tau_1 \rrbracket . Qf(U) \subseteq g(U)\}.$$

This establishes a type-indexed family of Galois connections $G \coloneqq \{\alpha_\tau \dashv \gamma_\tau : A(\tau) \to Q(\llbracket \tau \rrbracket)\}_{\tau \in Typ(B)}$.

Using the notations of Theorem 4.2, we obtain an abstract interpretation, that is, an oplax functor $\llbracket - \rrbracket_c^G : \lambda(\Pi) \to \mathbf{Pos}$ given by $\llbracket \tau \rrbracket_c^G \coloneqq A(\tau)$ and $\llbracket x : \tau \vdash P : \sigma \rrbracket_c^G \coloneqq \alpha_\sigma \circ Q(\llbracket x : \tau \vdash P : \sigma \rrbracket) \circ \gamma_\tau$. By Theorem 4.2, it comes with a concretization of interpretation γ from $\llbracket - \rrbracket_c^G$ to $Q \circ \llbracket - \rrbracket$, as well as an abstraction of interpretation α from $Q \circ \llbracket - \rrbracket$ to $\llbracket - \rrbracket_c^G$. The interpretation $\llbracket - \rrbracket_c^G$ is normal when the base type Galois connections are Galois insertions: indeed, it can be easily proved by induction on types that $\alpha_\tau \circ \gamma_\tau = \mathrm{id}$. However, it is not functorial.

To have an inductively defined semantics that over-approximates $\llbracket - \rrbracket_c^G$, we derive a new semantics of the lambda calculus that interprets each constant c by $\llbracket x : 1 \vdash c : typ(c) \rrbracket_c^G : 1_{\mathbf{Pos}} \to A(typ(c))$. Formally, we form a Π-structure $(A_0, \{\llbracket x : 1 \vdash c : typ(c) \rrbracket_c^G\}_{c \in O})$ in **Pos** and induce a Cartesian closed functor $(\!| - |\!) : \lambda(\Pi) \to \mathbf{Pos}$. This process is a lambda-calculus analogue of the derivation of $(\!| - |\!)$ from $\llbracket - \rrbracket_c^G$ for the while language in Section 4.2; in this example, assignment commands are replaced with constants in Π. The following theorem is in parallel with Theorem 4.3.

Theorem 5.1 *$(\!| - |\!)$ is a Cartesian closed functor, and $\llbracket - \rrbracket_c^G \leq (\!| - |\!)$.*

6 Related Works

In this paper, we study the construction of frameworks to describe semantic abstraction using categorical tools which are already used heavily to construct semantics of programming languages.

Abstract interpretation [9,10] was proposed as a very general framework to compare program semantics and has found applications in many areas of computer science such as semantics [8], program analysis and verification [9,5,16], program transformations [13,27], or security [17]. It is of very general scope as shown in [12], since it can be adapted to various program semantics, styles of abstraction relations (abstraction functions, concretization functions, Galois connections, or basic abstraction relations), abstract domains, and classes of approximation algorithms. Although it was remarked very early that it could also be formalized in category theory, most descriptions use set theory. Steffen et al [29] define both concrete and abstract semantics in a categorical framework in which they express soundness. Before their work, Panangaden and Mishra give a categorical formulation of abstract interpretation based on the concept called *quasi-functor* [26]. We leave the comparision of quasi-functors with oplax functors for future work. Sergey et al. [28,15] rely on monads to construct a control flow analysis of a small functional language, where various aspects of the program semantics and abstraction are described by monads. Our work uses categorical tools in a different way. First, we use a framework based on oplax functors and lax natural transformations in order to express and compare semantics. This is the right setting for embracing the induction of

semantics by Galois connections (Theorem 4.2), which is foundational in abstract interpretation. Second, we build upon a functorial decomposition to generalize the notion of collecting semantics, to expose a property component and a semantic component. According to [12] as *"the main utilization of the collecting semantics is to provide a sound and relatively complete proof method for the considered class of properties"*, however this presentation does not give a systematic way to construct one such semantics, thus one of our contributions is to provide a functorial interpretation for this notion. Another of our contributions is to bridge the gap between monadic tools to construct program semantics and their use in abstract interpretation.

This paper focuses on the forward property-based semantics. The other side of the story, namely the *backward* property-based semantics, has recently been categorically studied [19,2,31]. They point out that Dijkstra's *weakest precondition predicate transformer* (wppt for short), which is a typical backward property-based semantics, corresponds to functors of type $L^{op} \to \mathbf{Pos}$ (see wp^o in Example 3.9). This fact suggests that we may dualize this paper's story to develop abstract interpretations for backward property-based semantics. There, wppts expressed as functors would play the role of functorial collecting semantics.

7 Conclusion

We have introduced categorical structures that account for basic concepts and constructions in abstract interpretation. We have demonstrated that it can account for the abstraction of both imperative and functional programs in a unified manner. A future work is to explore various combinations of the data in the diagram (1). For the language category L, it would be interesting to take programming languages other than the while language and the simply typed lambda calculus, such as linear lambda calculus [4], Moggi's monadic metalanguage [25], and dependent type theories. For the property functor C, *regular fibrations* [20, Definition 4.2.1] are a rich source. The construction of (A, γ)-pairs is the central part of the development of abstract interpretation, and it remains to be seen if our categorical formalism provides new construction methods of (A, γ)-pairs.

Acknowledgement

The first and third authors were supported by ERATO HASUO Metamathematics for Systems Design Project (No. JPMJER1603), JST. The third author was also supported by JST, CREST Grant Number JPMJCR22M1. The second author was supported by the French ANR VeriAMOS project. The authors are grateful to James Haydon for his critical reading of the manuscript, and implementation proposals and suggestions of new directions during discussions. The authors are also grateful to Ichiro Hasuo for insightful comments and suggestions.

References

[1] Adámek, J., J. Rosický, E. Vitale and F. W. Lawvere, *Algebraic Theories: A Categorical Introduction to General Algebra*, Cambridge Tracts in Mathematics, Cambridge University Press (2010).

[2] Aguirre, A. and S. Katsumata, *Weakest preconditions in fibrations*, in: P. Johann, editor, *Proceedings of the 36th Conference on the Mathematical Foundations of Programming Semantics, MFPS 2020, Online, October 1, 2020*, volume 352 of *Electronic Notes in Theoretical Computer Science*, pages 5–27, Elsevier (2020). https://doi.org/10.1016/j.entcs.2020.09.002

[3] Assaf, M., D. A. Naumann, J. Signoles, E. Totel and F. Tronel, *Hypercollecting semantics and its application to static analysis of information flow*, in: *Symposium on Principles of Programming Languages (POPL)*, POPL '17, pages 874–887, Association for Computing Machinery, New York, NY, USA (2017), ISBN 9781450346603. https://doi.org/10.1145/3009837.3009889

[4] Benton, N., G. Bierman, V. de Paiva and M. Hyland, *Linear λ-calculus and categorical models revisited*, in: E. Börger, G. Jäger, H. Kleine Büning, S. Martini and M. M. Richter, editors, *Computer Science Logic*, pages 61–84, Springer Berlin Heidelberg, Berlin, Heidelberg (1993), ISBN 978-3-540-47890-4. https://doi.org/10.1007/3-540-56992-8_6

[5] Blanchet, B., P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux and X. Rival, *A static analyzer for large safety-critical software*, in: R. Cytron and R. Gupta, editors, *Conference on Programming Languages Design and*

Implementation (PLDI), pages 196–207, ACM (2003).
https://doi.org/10.1145/780822.781153

[6] Clarkson, M. R. and F. B. Schneider, *Hyperproperties*, Journal of Computer Security **18**, pages 1157–1210 (2010).
https://doi.org/10.3233/JCS-2009-0393

[7] Cortesi, A., G. Costantini and P. Ferrara, *A survey on product operators in abstract interpretation*, Electronic Proceedings in Theoretical Computer Science **129**, pages 325–336 (2013).
https://doi.org/10.4204/eptcs.129.19

[8] Cousot, P., *Constructive design of a hierarchy of semantics of a transition system by abstract interpretation*, in: *Conference on Mathematical Foundations of Progamming Semantics (MFPS)*, volume 6 of *Electronic Notes in Theoretical Computer Science (ENTCS)*, pages 77–102, Elsevier (1997).
https://doi.org/10.1016/S1571-0661(05)80168-9

[9] Cousot, P. and R. Cousot, *Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints*, in: *Symposium on Principles of Programming Languages (POPL)*, page 238–252, ACM (1977).
https://doi.org/10.1145/512950.512973

[10] Cousot, P. and R. Cousot, *Systematic design of program analysis frameworks*, in: *Symposium on Principles of Programming Languages (POPL)*, page 269–282, ACM (1979).
https://doi.org/10.1145/567752.567778

[11] Cousot, P. and R. Cousot, *Abstract interpretation and application to logic programs*, Journal of Logic Programming **13**, pages 103–179 (1992).
https://doi.org/10.1016/0743-1066(92)90030-7

[12] Cousot, P. and R. Cousot, *Abstract interpretation frameworks*, Journal of Logic and Computation **2**, pages 511–547 (1992).
https://doi.org/10.1093/logcom/2.4.511

[13] Cousot, P. and R. Cousot, *Systematic design of program transformation frameworks by abstract interpretation*, in: *Symposium on Principles of Programming Languages (POPL)*, pages 178–190, ACM (2002).
https://doi.org/10.1145/565816.503290

[14] Crole, R. L., *Categories for Types*, Cambridge University Press (1994).
https://doi.org/10.1017/CBO9781139172707

[15] Darais, D., M. Might and D. V. Horn, *Galois transformers and modular abstract interpreters: reusable metatheory for program analysis*, in: *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 552–571, ACM (2015).
https://doi.org/10.1145/2814270.2814308

[16] Distefano, D., M. Fähndrich, F. Logozzo and P. W. O'Hearn, *Scaling static analyses at facebook*, Communications of the ACM **62**, pages 62–70 (2019).
https://doi.org/10.1145/3338112

[17] Giacobazzi, R. and I. Mastroeni, *Abstract non-interference: parameterizing non-interference by abstract interpretation*, in: *Symposium on Principles of Programming Languages (POPL)*, pages 186–197, ACM (2004).
https://doi.org/10.1145/982962.964017

[18] Goncharov, S. and L. Schröder, *A relatively complete generic hoare logic for order-enriched effects*, in: *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28, 2013*, pages 273–282, IEEE Computer Society (2013), ISBN 978-1-4799-0413-6.
https://doi.org/10.1109/LICS.2013.33

[19] Hasuo, I., *Generic weakest precondition semantics from monads enriched with order*, Theor. Comput. Sci. **604**, pages 2–29 (2015).
https://doi.org/10.1016/j.tcs.2015.03.047

[20] Jacobs, B., *Categorical Logic and Type Theory*, volume 141 of *Studies in Logic and the Foundations of Mathematics*, Elsevier (2000).

[21] Kildall, G., *A unified approach to global program optimization*, in: *Symposium on Principles of Programming Languages (POPL)*, pages 194–206, ACM (1973).
https://doi.org/10.1145/512927.512945

[22] Lambek, J. and P. J. Scott, *Introduction to Higher-Order Categorical Logic*, Cambridge Studies in Advanced Mathematics, Cambridge University Press (1988).

[23] Lawvere, W., *Functorial Semantics of Algebraic Theories*, Ph.D. thesis, Columbia University (1963).

[24] MacLane, S., *Categories for the Working Mathematician (Second Edition)*, volume 5 of *Graduate Texts in Mathematics*, Springer (1998).

[25] Moggi, E., *Notions of computation and monads*, Information and Computation **93**, pages 55–92 (1991). https://doi.org/10.1016/0890-5401(91)90052-4

[26] Panangaden, P. and P. Mishra, *A category theoretic formalism for abstract interpretation*, Technical Report UUCS-84-005, Department of Computer Science, the University of Utah (1984).

[27] Rival, X., *Symbolic transfer function-based approaches to certified compilation*, in: N. D. Jones and X. Leroy, editors, *Symposium on Principles of Programming Languages (POPL)*, pages 1–13, ACM (2004). https://doi.org/10.1145/964001.964002

[28] Sergey, I., D. Devriese, M. Might, J. Midtgaard, D. Darais, D. Clarke and F. Piessens, *Monadic abstract interpreters*, in: *Conference on Programming Languages Design and Implementation (PLDI)*, pages 399–410, ACM (2013). https://doi.org/10.1145/2491956.2491979

[29] Steffen, B., C. B. Jay and M. Mendler, *Compositional characterization of observable program properties*, Theoretical Informatics and Applications **26**, pages 403–424 (1992).

[30] Venet, A., *Abstract cofibered domains: Application to the alias analysis of untyped programs*, in: *Static Analysis Symposium (SAS)*, volume 1145 of *Lecture Notes in Computer Science*, pages 366–382, Springer (1996).

[31] Wolter, U. E., A. R. Martini and E. H. Häusler, *Indexed and fibered structures for partial and total correctness assertions*, Mathematical Structures in Computer Science pages 1–31 (2022). https://doi.org/10.1017/S0960129522000275

[32] Zhang, L. and B. L. Kaminski, *Quantitative strongest post: A calculus for reasoning about the flow of quantitative information*, Proc. ACM Program. Lang. **6** (2022). https://doi.org/10.1145/3527331