A Categorical Framework for Program Semantics and Semantic Abstraction (Extended Abstract)

Shin-ya Katsumata Xavier Rival Jérémy Dubut

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.

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.

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 [9]; the calculus modulo $\beta\eta$-equality is presented as a Cartesian closed category, and its semantics in a Cartesian

---

* プログラム意味論とその抽象化のための圏論的枠組み
This is an excerpt of the introduction section of the paper accepted by MFPS 2023. This excerpt should not be regarded as a publication.

勝股 審也, 国立情報学研究所, National Institute of Informatics.
ジェレミー デュブ, 産業総合研究所, Advanced Institute of Science and Technology.
ザビエル リバル, フランス国立情報学自動制御研究所, Institut National de Recherche en Informatique et en Automatique.

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* [10] [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 [11].

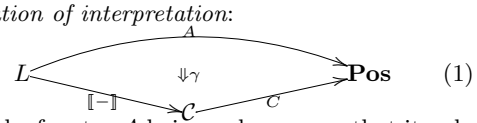
Abstract interpretation [4] provides a framework to compare program semantics of varying levels of expressiveness, and to derive sound approximations of program semantics, based on a given abstraction relation. It has been used to describe relationships across program semantics [3], program analysis [4] [2] [8], program transformations [7], 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 [4] [6], 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 [13] 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 [14] 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 [5]. More recently, Sergey et al. [12] 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 *con-*

cretization 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 [13]. We adopt the same approach, and further bring some basic concepts that are not covered in [13] 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 [13], we formalize abstract interpretation in a functorial semantics framework. The new ingredients from [13] are the following:

1. We show that interpretations (oplax functors) are closed under the *induction operation* by Galois connections. This induction also comes with a *concretizations of interpretations*, formulated as lax natural transformations.
2. We give a categorical formulation of *collecting semantics*, 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.
3. We present two examples of developments of abstract interpretations, one for a while language over generic computational effects, and the other for the simply typed lambda calculus. 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. This semantics is a generalization of the strongest postcondition semantics introduced in [15].

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.

参 考 文 献

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

[2] Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., and Rival, X.: A static analyzer for large safety-critical software, *Conference on Programming Languages Design and Implementation (PLDI)*, Cytron, R. and Gupta, R.(eds.), ACM, 2003, pp. 196–207.

[3] Cousot, P.: Constructive design of a hierarchy of semantics of a transition system by abstract interpretation, *Conference on Mathematical Foundations of Progamming Semantics (MFPS)*, Electronic Notes in Theoretical Computer Science (ENTCS), Vol. 6, Elsevier, 1997, pp. 77–102.

[4] Cousot, P. and Cousot, R.: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints, *Symposium on Principles of Programming Languages (POPL)*, ACM, 1977.

[5] Cousot, P. and Cousot, R.: Systematic Design of Program Analysis Frameworks, *Symposium on Principles of Programming Languages (POPL)*, ACM, 1979.

[6] Cousot, P. and Cousot, R.: Abstract Interpretation Frameworks, *Journal of Logic and Computation*, Vol. 2, No. 4(1992), pp. 511–547.

[7] Cousot, P. and Cousot, R.: Systematic design of program transformation frameworks by abstract interpretation, *Symposium on Principles of Programming Languages (POPL)*, ACM, 2002, pp. 178–190.

[8] Distefano, D., Fähndrich, M., Logozzo, F., and O'Hearn, P. W.: Scaling static analyses at Facebook, *Communications of the ACM*, Vol. 62, No. 8(2019), pp. 62–70.

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

[10] Lawvere, W.: *Functorial Semantics of Algebraic Theories*, PhD Thesis, Columbia University, 1963.

[11] Moggi, E.: Notions of Computation and Monads, *Information and Computation*, Vol. 93, No. 1(1991), pp. 55–92.

[12] Sergey, I., Devriese, D., Might, M., Midtgaard, J., Darais, D., Clarke, D., and Piessens, F.: Monadic abstract interpreters, *Conference on Programming Languages Design and Implementation (PLDI)*, ACM, 2013, pp. 399–410.

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

[14] Venet, A.: Abstract Cofibered Domains: Application to the Alias Analysis of Untyped Programs, *Static Analysis Symposium (SAS)*, Lecture Notes in Computer Science, Vol. 1145, Springer, 1996, pp. 366–382.

[15] Zhang, L. and Kaminski, B. L.: Quantitative Strongest Post: A Calculus for Reasoning about the Flow of Quantitative Information, *Proc. ACM Program. Lang.*, Vol. 6, No. OOPSLA1(2022).