Cumulative Abstract Semantics

CADE LUEKER, University of Colorado Boulder, USA
ANDREW FOX, University of Colorado Boulder, USA

TODO

1 Introduction(both)

Desired contributions of paper:
- we introduce cumulative semantics as an idea
- we give a formalization for: concrete, collecting, and abstract interpreter in the context of cumulative semantics
- a comparison between effect and typeclass based implementation of cumulative semantics
- soundness proofs?

2 Overview(both)

Often when one is crafting an analysis, they begin with a fixed target language in mind, and analysis they would like to apply to that language. We begin this example in a different way: with a fixed analysis we would like to do on a growing language. This analysis will be a concrete interpretation of a simple language. To start this language we will introduce the idea of numbers:

$$\begin{array}{lcll} \textit{numbers} & n & \in & \mathbb{Z} \\ \textit{expressions} & \exp & := & \texttt{cst}(n) \mid e_1 + e_2 \end{array}$$

Evaluation of this language can be done trivially in any number of ways, and we will choose one that is most convenient for our purposes. We will define a judgment form $[\![\textit{expr}]\!]$

3 Technical(both, maybe more andrew)

talk through formalization of it in here, steo through ok this is the step 0 unsubstantiated semantics, this is the collecting, this is the conrcete, this is the symbolic. Look at how nice they play together.

3.1 Standard abstract interpretation recipe (review)

3.2 New recipe with these semantics

3.3 concrete

3.4 collecting

3.5 abstract(backwards)?

4 Evaluation

Two case studies:

4.1 Effect based (cade)

In here show concrete and backwards symbolic in effekt/koka

4.2 Typeclass based (andrew)

In here show concrete and backwards symbolic in lean

Authors' Contact Information: Cade Lueker, University of Colorado Boulder, Boulder, Colorado, USA, cade.lueker@colorado.edu; Andrew Fox, Andrew.Fox@colorado.edu, University of Colorado Boulder, Boulder, Colorado, USA.

5 Conclusion(both)

compare the two methodologies, talk about how great both are and the future of both: for the effect stuff (assuming it's easier to build) we can make a larger repo for many languages and accomplish a lot. For the lean one talk about how you get to reason about it for free.

6 Related Works(both)

Van Horn Abstracting abstract machines. [2]: Sergey built on this idea, it seems to allow for the isolation of more complex ideas, allowing for their simplification and independent reasoning. They accomplish this by abstracting the store (and making it finite), turning an infinite execution tree into a finite state graph.

Sergey Monadic Abstract Interpreters. [7]: This work seems to mostly be based off of a previous work (abstracting abstract machines) but also using a monad to capture state. Their essential claim seems to be similar to ours in that they are making the semantics orthogonal(and reusable) across different interpreters with the monadic approach. By using a monadic state you delay any choices in your interpreter until you actually assemble it. In my monadic attempt we take the idea of using a monadic state to track it easily through everything, keeping its representation orthogonal to what we want. The monadic implementation of intro and elim is similar to this paper, but slightly different because we retain a more precise control over control flow. Also, our version is done in a more modern and dependently typed language leaving room to reason and prove things if we wanted to.

Michelland Monadic Modular Verification. [5]: They advanced the usage of monadic modularity with their ITree-based framework for abstract interpretation, composing state and control flow monads for sound meta-theory in Roq. Their approach supports modular analyses (e.g., binding-time analysis) but requires complex transformer stacks. Their work is also done in a dependently type language like ours.

Keidel Sturdy. [3, 4]: The goal of this work is automatic soundness proofs for components, so not centrally aligned with what we are going for. Their approach provides soundness guarantees for free but is fixed to non-relational domains and requires an implementation of a monolithic, generic interpreter for every language. It accomplishes modularity with arrow transformers which are generally more rigid than effects or monads with control flow manipulation.

Reinder Modular Effect Interpreter. [6]: This short paper introduced the idea to make an effect based interpreter in theory, and played with a small toy example of it.

Bunkenburg Making a Curry Interpreter. [1]: They started using effect handlers embedded in Haskell to build a modular interpreter, one of the first of its kind. Its focus was on concrete interpretation of the Curry language, but Curry already supports non-determinism which is the same as non determinism in abstract interpretation. This demonstrated the usefulness of effects for control flow manipulation, which we use and extend to more generic domains in our work.

References

[1] Niels Bunkenburg and Nicolas Wu. 2024. Making a Curry Interpreter using Effects and Handlers. In *Proceedings of the 17th ACM SIGPLAN International Haskell Symposium, Haskell 2024, Milan, Italy, September 6-7, 2024*, Niki Vazou and J. Garrett Morris (Eds.). ACM, 68–82. https://doi.org/10.1145/3677999.3678279

[2] David Van Horn and Matthew Might. 2010. Abstracting abstract machines. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, Paul Hudak and Stephanie Weirich (Eds.). ACM, 51–62. https://doi.org/10.1145/1863543.1863553

[3] Sven Keidel and Sebastian Erdweg. 2019. Sound and reusable components for abstract interpretation. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 176:1–176:28. https://doi.org/10.1145/3360602

[4] Sven Keidel, Casper Bach Poulsen, and Sebastian Erdweg. 2018. Compositional soundness proofs of abstract interpreters. *Proc. ACM Program. Lang.* 2, ICFP (2018), 72:1–72:26. https://doi.org/10.1145/3236767

[5] Sébastien Michelland, Yannick Zakowski, and Laure Gonnord. 2024. Abstract Interpreters: A Monadic Approach to Modular Verification. *Proc. ACM Program. Lang.* 8, ICFP (2024), 602–629. https://doi.org/10.1145/3674646

[6] Jaro S. Reinders. 2023. Towards Modular Compilation Using Higher-Order Effects. In *Eelco Visser Commemorative Symposium, EVCS 2023, April 5, 2023, Delft, The Netherlands (OASIcs, Vol. 109)*, Ralf Lämmel, Peter D. Mosses, and Friedrich Steimann (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 22:1–22:9. https://doi.org/10.4230/OASICS.EVCS.2023.22

[7] Ilya Sergey, Dominique Devriese, Matthew Might, Jan Midtgaard, David Darais, Dave Clarke, and Frank Piessens. 2013. Monadic abstract interpreters. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 399–410. https://doi.org/10.1145/2491956.2491979