Sound and Complete Refinement for SSA with Substructural Effects

JAD GHALAYINI and NEEL KRISHNASWAMI

CCS Concepts: • **Theory of computation → Denotational semantics**; **Categorical semantics**; **Type theory**.

Additional Key Words and Phrases: SSA, Categorical Semantics, Elgot Structure, Effectful Category

ACM Reference Format:
Jad Ghalayini and Neel Krishnaswami. 2025. Sound and Complete Refinement for SSA with Substructural Effects. 1, 1 (March 2025), 3 pages. https://doi.org/XXXXXXX.XXXXXXX

Compiler optimizations are fundamentally directed rewrites. Most production compilers use the *static single assignment*, or SSA, as their intermediate representation, with optimization passes consisting of transformations from one *effectful* SSA program into another. This immediately leads to a correctness question: is the output program a *refinement* of the input program?

Depending on what style semantics we assign our program, we can attempt to answer this question in two different ways. Using operational semantics, we can verify transformations by constructing explicit simulation proofs, demonstrating that optimized programs replicate the behavior of their originals. A notable example is Zhao et al. [5]'s work on Vellvm, which introduces multiple operational semantics for LLVM and establishes refinements among them. For instance, verifying the SoftBound transformation [4] involved defining a bespoke operational semantics tailored specifically to SoftBound and then proving simulation relative to the original semantics.

In contrast, denotational semantics allow verification by establishing semantic equivalence or refinement of the denotations of the original and transformed programs. Even complex side effects, such as those arising in weak memory models, can often be elegantly captured using monadic frameworks. Examples include monadic Brookes-style semantics for release-acquire (Dvir et al. [1]) and TSO (Jagadeesan et al. [3]) memory models. Compositionality means that denotational semantics can often be used to reason about optimizations *inductively*, as well as effectively reason about small program transformations such as peephole rewrites.

Simulation proofs in operational semantics are often labor-intensive and lack reusability, necessitating new bespoke proofs for each optimization or additional side effect. Conversely, denotational semantics demands significant initial investment to construct comprehensive mathematical models, with resulting proofs that typically remain tightly coupled to the details of a specific model. Using denotational semantics also requires us to figure out how to break down a program into components, and compositionally assign a semantics to those components; this has recently been achieved for SSA by Ghalayini and Krishnaswami [2].

However, many practical compiler optimizations, such as loop hoisting, `mem2reg`, and sparse conditional constant propagation, seem not to depend upon the specific details of the model, instead

Authors' Contact Information: Jad Ghalayini, jeg74@cl.cam.ac.uk; Neel Krishnaswami, nk480@cl.cam.ac.uk.

ACM XXXX-XXXX/2025/3-ART
https://doi.org/XXXXXXX.XXXXXXX

relying on high level information such as which operations are pure/constant; for example, Dvir et al. [1] note that, since they use a monadic semantics, they have access to a huge amount of generic machinery to deal with, e.g., higher-order programs, "for free". In particular, we observe that we often only require information about:

(1) How different effects commute; for example, printing commutes with (potential) UB, but not with (potential) nontermination. Sometimes, this is a *refinement*, rather than an *equation*: since UB followed by nontermination is UB (which can be optimized to nontermination), but nontermination followed by UB is nontermination (which cannot be optimized to UB)
(2) Peephole rewrties on effectful operations relevant to our optimization; for example, a `mem2reg` optimization may not care about details of our memory model except for facts like store $a\ b$; get $a \twoheadrightarrow$ let $x = b$; store $a\ x$; x, where $\twoheadrightarrow$ indicates refinement. In particular, we are interested in
(3) The *linearity* of different effects. For example, we can't duplicate or discard a printing operation, so it is *linear*. On the other hand, an effect like nontermination can be duplicated (e.g. let x = maybeloop; $(x, x) \approx$ (maybeloop, maybeloop)), i.e. is *relevant*, while an effect like nondeterminism can be discarded (e.g. let $\cdot$ = nondet; $5 \approx 5$), i.e. is *affine*. Again, linearity is sometimes a refinement rather than an equation; for example, (nondet, nondet) $\twoheadrightarrow$ let x = nondet; (x, x), making nondeterminism *fusable*, but not relevant.

In this work, we introduce λ_{iter}, a simple calculus for reasoning about program refinement given an effect system and information about (directed) commutativity, linearity, and an arbitrary set of peephole rewrites. We then proceed to:

- Show λ_{iter} is in fact equivalent to (substructural) SSA, and explore how this can be used to reason about compiler optimization passes
- Provide a denotational semantics for λ_{iter}, which we prove is sound and complete
- Give some interesting models of λ_{iter}, supporting effects such as weak memory and complex type-system features such as separation logic

We hope to use this work as a starting point to explore the utility of high-level algebraic approaches to building model and effect-generic optimizations and analysis passes for effectful programs, as well as to verify that new models are compatible with standard optimizations.

References

[1] Yotam Dvir, Ohad Kammar, and Ori Lahav. 2024. A Denotational Approach to Release/Acquire Concurrency. In *Programming Languages and Systems - 33rd European Symposium on Programming, ESOP 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 14577)*, Stephanie Weirich (Ed.). Springer, Luxembourg City, Luxembourg, 121–149. https://doi.org/10.1007/978-3-031-57267-8_5

[2] Jad Elkhaleq Ghalayini and Neel Krishnaswami. 2024. The Denotational Semantics of SSA. *arXiv e-prints*, Article arXiv:2411.09347 (Nov. 2024), arXiv:2411.09347 pages. https://doi.org/10.48550/arXiv.2411.09347 arXiv:2411.09347 [cs.PL]

[3] Radha Jagadeesan, Gustavo Petri, and James Riely. 2012. Brookes Is Relaxed, Almost!. In *Foundations of Software Science and Computational Structures - 15th International Conference, FOSSACS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7213)*, Lars Birkedal (Ed.). Springer, Tallinn, Estonia, 180–194. https://doi.org/10.1007/978-3-642-28729-9_12

[4] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. 2009. SoftBound: highly compatible and complete spatial memory safety for c. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, Michael Hind and Amer Diwan (Eds.). ACM, 245–258. https://doi.org/10.1145/1542476.1542504

[5] Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2012. Formalizing the LLVM intermediate representation for verified program transformations. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, John Field and Michael Hicks (Eds.). ACM, Philadelphia, Pennsylvania, 427–440. https://doi.org/10.1145/2103656.2103709