Equational Proofs of Optimizations with Interaction Trees

LUCAS SILVER, University of Pennsylvania, USA
IRENE YOON, University of Pennsylvania, USA
YANNICK ZAKOWSKI, University of Pennsylvania, USA
STEVE ZDANCEWIC, University of Pennsylvania, USA

1 INTRODUCTION

Proofs are hard to modularize. This talk will describe how the use of equational reasoning, applied to different interpretations of *interaction trees* [Xia et al. 2020], leads to modular proofs about program semantics. Interaction trees take heavy influence from algebraic effects and handlers [Plotkin and Pretnar 2013] to obtain modular reasoning principles; are used as a denotational semantic domain for compositionality; and come equipped with combinators supplied with a rich equational theory allowing one to reason up-to weak bisimulation. We present a case study of how using ITrees leads to a crisp separation of concerns and ease of proof-writing in verifying a simple compiler optimization proof.

2 INTERACTION TREES

```
CoInductive itree E A : Type
  | Tau : itree E A → itree E A
  | Ret : A → itree E A
  | Vis : ∀{R} (e: E R),
     (R → itree E A) → itree E A
```

Fig. 1. Simplified Definition of ITree.

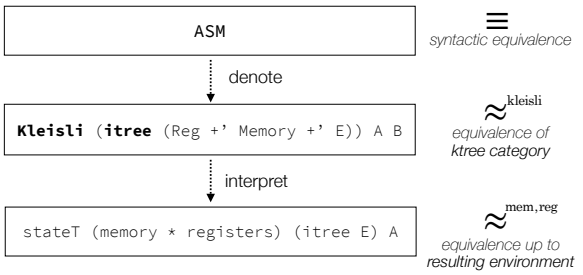

Fig. 2. Interpretation levels and equivalences used at each level.

Interaction Trees (*ITrees*) are a data structure for representing the computational behavior of programs that interact with their environments. They are implemented as a coinductive type, indexed by a return type `R` and type family of events `E`, `itree E R`, with three constructors. `Ret` models a pure computation returning the result `r`. `Tau` represents a silent internal step of the computation. `Vis` contains a *visible* external event of type `E R`, and a continuation of type `R → itree E A` that takes a response from the environment. Visible events represent an uninterpreted effect that needs to be passed to the environment in order to be interpreted. To give semantics to events, an *event handler* of type `∀ A, E A → M A` may be defined for some monad `M`. This *event handler* gets lifted to interpret the entire ITree into the target monad by the `interp` function. ITrees come equipped with an equivalent up to *tau* (`eutt`) relation that generalizes the natural notion of bisimulation by allowing a finite number of `Tau` constructors to be ignored. This relation relates trees that emit the exact same *visible* events and return equal values.

Events and handlers induce levels of interpretations of a program, where distinct notions of equivalences may be used for each level. We use a denotation and interpretation of ASM as an example, as seen in Figure 2. Equality on the original ASM program is on syntactic equivalence, while the denoted program is a representation of an open program through ITrees (referred to as the ktree category, the Kleisli category of ITrees). This is interpreted into a state monad, where

the state is the mapping of values in memory and register locations. The equivalence on ITrees (`eutt`) can be parameterized by a relation on return types: it is therefore a convenient tool to define modular notions of equivalences at each level of interpretation that allow for arbitrary bisimulation relations over computed states. Section 2 will give an overview of a peephole optimization and how using the equational theory of ITrees leads to its modular verification.

3 CASE STUDY: PEEPHOLE OPTIMIZATION

Using ITrees, we can create denotational semantics for a wide variety of languages. With denotational semantics, it is relatively easy to define notions of program equivalence for use in compiler and optimization correctness. In this talk, we will discuss the semantics for a simple, model assembly language ASM, and an equational proof of correctness of an optimization over ASM. We discuss here specifically a simple set of peephole optimizations, a common optimization technique that replaces a sequence of instructions to a semantically equivalent and more performant sequence of instructions, but the technique we present is general.

ASM programs are represented as control flow subgraphs linking basic blocks by branching labels and allowing for open programs to expose their interface. They are denoted as ITrees over an event type `AsmEv` for memory and register operations.

This inert denotation gets interpreted through nested state monad transformers, implementing respectively the local registers and the main memory. Uninterpreted denotations of ASM programs, of type `itree AsmEv R`, are interpreted into the type `regs → mem → itree EmptyE (regs * mem * R)`. Albeit possibly divergent, these computations are now executable given initial register and heap states, and return final register and heap states, as well as a return value of type `R`. Then we can consider two such programs to be equivalent if, given initial register and memory values related by some relation `sim`, they return trees related by `eutt` parameterized by `sim`. Our meta-theory ensures that the equational theory provided by itrees is preserved through this interpretation.

In this talk, we will discuss how this denotation through ITrees allows us to use their equational theory and rewriting techniques to prove the correctness of peephole optimizations. Our proof[1] states that for any given peephole optimization, the optimized program and initial program have equivalent denotations (i.e. the mapping of values in registers and heap coincide with each other). The two-phased denotation, supported by the equational theory of ITrees, allows for a separation of concern when establishing the behavioral equivalence between the original program and its optimized counterpart: the reasoning about the control-flow is resolved through the itrees combinators, while the semantic reasoning about state is delayed after interpretation. The proofs only use basic rewriting using the equational theory library of ITrees and require no explicit coinduction, despite being termination sensitive.

4 DISCUSSION

Monads, algebraic effects and handlers [Bauer and Pretnar 2015; Plotkin and Pretnar 2013], and effects in type theory [Swierstra 2009] are well-studied subjects that have been used to reason about effectful behavior in pure functional settings. Interaction Trees lift these methodologies to build an extensible denotational semantics for possibly diverging programs in a termination-sensitive interactive theorem prover. It is future work to explore the extensibility of ITrees, and recent efforts in the Vellvm[2] [Zhao et al. 2012], a verified LLVM [Lattner and Adve 2004] framework, is using Interaction Trees to give a modular semantics to the LLVM compiler infrastructure.

[1] Available on GitHub: https://github.com/DeepSpec/InteractionTrees/tree/master/tutorial
[2] https://www.cis.upenn.edu/~stevez/vellvm/

REFERENCES

Andrej Bauer and Matija Pretnar. 2015. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming* 84, 1 (Jan 2015), 108–123.

Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation *(CGO '04)*. IEEE Computer Society, USA, 75.

Gordon D Plotkin and Matija Pretnar. 2013. Handling Algebraic Effects. *Logical Methods in Computer Science* 9, 4 (Dec. 2013). https://doi.org/10.2168/LMCS-9(4:23)2013

Wouter Swierstra. 2009. *A functional specification of effects*. Ph.D. Dissertation. University of Nottingham, UK. http://eprints.nottingham.ac.uk/10779/

Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2020. Interaction Trees: Representing Recursive and Impure Programs in Coq. *Principles of Programming Languages* 4 (Jan. 2020). https://doi.org/10.1145/3371119

Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. 2012. Formalizing the LLVM Intermediate Representation for Verified Program Transformations. (2012), 427–440. https://doi.org/10.1145/2103656.2103709