Formal Verification of Monadic Computations

Steve Zdancewic

June 2024

1 Interaction Trees : A Coinductive Version of the Freer Monad

Computation can either return a value of type R, perform a internal step of computation, or tirgger an external event e expecting a certain answer of type X. In this final case, we must continue with a continuation that takes in things of type X.

1.1 How are ITrees executable?

They are just coinductive data structres. Meaning that they are lazily evaluated, and we can unfold one layer by forcing the tree. An interpreter is then something that steps through the tree by iteratively peeling back one lazy constructor

Note: That these being "executable" means that we can extract it to an OCaml program

2 First Examples of ITree Computations

We can choose what sorts of events we wish to include, as a first pass lets include IO events

2.1 Silent Divergence

We can forever take τ steps, and thus never emit an event. This program has no observable behavior, and is thus in the bottom of a lattice of programs ordered by their observable behavior

2.2 Failure

We can model failure by having nodes that expect continuations that expect terms of the empty type

2.3 Sequential Composition

Find all leaves of a tree and replace return nodes in a tree with continutation that return new trees. Stitch together existing continuation leaves by composition with the incoming continuations.

This sequential composition provides the bind for the monad structure on ITrees.

3 Factorial via ITrees

Steve's notes show how to encode iterative programs, and I won't go into regurgitating all of that. However, I think that it is important to reflect on how these programs work. The structure of the ITrees allows coinductive reasoning of potentially non-terminating programs. Even though that the programs he is writing happen to terminate, that is not a given.

4 Intepretations

Interpretations (or interpreters) are just monadic folds over the ITree.

5 Denotational Semantics for Imp

Represent the read and write interactions of Imp as read and write events. We are weriting a very similar interpreter as we did yesterday, with an approach like in datatype a la carte, but now where we output ITrees.

I wish I had more to say here but its hard to write anything without just echoing his slides

6 Bisimulation

Can call two ITrees equivalent, *strongly bisimilar*, if they are syntactically equal, node-for-node, when unrolled.

We may also introduce a notion of equivalence for *weak bisimilarity*, where we talk about equality up to ignoring τ nodes. To be more precise, in Steve's words, "ignoring *inductively many* τ's". That is, we can ignore finitely many τ nodes, but we cannot delete or impute infinitely many τ's to the point where you affect the divergence of a path through the tree.

Both weak and strong bisimulation respect the monad laws. However, iteration is better behaved with respect to weak bisimilarity.

7 Equivalence for Imp Programs

We have syntactically written Imp programs, and we represent them as ITrees. We can think of this as a very weak sense of compilation down to ITrees. This gives a stack of progressively lower level representations. Moreover, this compilation is sound in the sense that equivalence of higher level representations implies equivalence of their lower level analogues.

We can prove the soundness of syntactically rewriting rules — justified by proving a lemma of having the same interpretation into ITrees — and then we can write proof search tactics that automatically apply these rewriting rules.

Similarly, you can build optimizing program transformations and then prove them correct via the tactics.