Telescopic Proof Refinement

JANUARY 02, 2018 DARRYL MCADAMS 14 MIN READ

In the third post in this series (part 1, part 2) on proof refinement, I'm going to show you how to properly handle bidirectionality in an elegant way. The technique we'll use is the replacement of lists and functions with a data structure called a telescope. This post will use Haskell exclusively, because of the limitations of JavaScript in presenting these things elegantly.

I've put together repl.it REPLs for this blog post so you can play around with the code. You can find them here: Addition 1, Addition 2, Addition 3, Lambda Calculus.

Consider the type that represented a successful decomposition of a problem judgment into subproblems, when working in the proof system for addition: `([Judgment], [Nat] -> Nat)`. The list of judgments represents the subproblems, and the function represents how to compute the result of the main problem from the results of the subproblems. This was problematic to generalize though, because it meant that all of the subproblems had to be independent. You couldn't use the result of solving an earlier subproblem to state what a later problem was. Information flowed strictly from subproblems out to main problems, never into other subproblems.

This of course was because the subproblems were all given at the same time, and the results were all simultaneous arguments to the function that computed the main result. For instance, we might have a decomposition that looked like `([j0,j1,j2], f)` where `f = \[r0,r1,r2] -> ...`, and we would solve these basically as `f [solve j0, solve j1, solve j2]`. What could we do to make it possible to have dependence of later problems are earlier results, though? Well, we could produce subproblems and consume results one at a time. In fact, instead

of the above pair, why not have `(j0, \r0 -> (j1, \r1 -> (r2, \r2 -> ...)))`. This has the type `(Judgment, Nat -> (Judgment, Nat -> (Judgment, Nat -> Nat)))`, which is specific to the case where the list of subproblems has exactly three subproblems in it. This doesn't generalize well, but we can notice the obvious recursive pattern and instead define

```
data Problems = Done Nat
              | SubProblem Judgment (Nat -> Problems)
```

Here, we're either done, and we have a resulting number, or we have a subproblem to solve, and a way of getting from the result of solving it to some more problems. Now of course, decomposing actually produced a Maybe `([Judgment], [Nat] -> Nat)`, so we really ought to define this type to account for the Nothing case as well:

```
data Problems = Fail
              | Done Nat
              | SubProblem Judgment (Nat -> Problems)
```

Our decompositions now will look mostly the same, but slightly different:

```
decomposePlus12 :: Nat -> Nat -> Problems
decomposePlus12 Zero     y = Done y
decomposePlus12 (Suc x) y = SubProblem (Plus12 x y) (\z -> Done (Suc z))

decomposePlus13 :: Nat -> Nat -> Problems
decomposePlus13 Zero z = Done z
decomposePlus13 (Suc x) (Suc z) = SubProblem (Plus13 x z) (\z -> Done z)
decomposePlus13 _ _ = Fail

decompose :: Judgment -> Problems
decompose (Plus12 x y) = decomposePlus12 x y
decompose (Plus13 x z) = decomposePlus13 x z
```

Finding a proof is pretty easy now too, because we can just define it in in terms of a second function that handles problems more generally. Dropping the reconstruction of a proof tree, we have:

```
findProof :: Judgment -> Maybe Nat
findProof j = solveProblems (decompose j)

solveProblems :: Problems -> Maybe Nat
solveProblems Fail = Nothing
solveProblems (Done x) = return x
solveProblems (SubProblem j f) =
  do x <- findProof j
     solveProblems (f x)
```

The interesting thing here is how we solve problems. If we fail, well, we've failed, so there's nothing to return. If we've finished, we've finished and so there's a number to return. But what if we have a subproblem? Well, we simply find a proof for it, computing the result as x, and then use the result of that to get more problems to solve, and solve those.

Generalizing

Having established the general shape of this approach, we can now move on to generalizing the pattern involved. The first move we'll make is to observe that we might want to generalize the type of judgments to index for the type of their result. After all, we might also want to include predicates in the class of possible judgments, where there are no useful return values at all, just (). So we can generalize Judgment, and in term, Problems, like so:

```
data Judgment r where
  Plus12 :: Nat -> Nat -> Judgment Nat
  Plus13 :: Nat -> Nat -> Judgment Nat

data Problems r where
  Fail :: Problems r
  Done :: r -> Problems r
  SubProblem :: Judgment s -> (s -> Problems r) -> Problems r
```

As soon as we do this, we discover that Problems looks an awful lot like a monad, and indeed, it is!

```
instance Functor Problems where
  fmap f Fail = Fail
  fmap f (Done x) = Done (f x)
  fmap f (SubProblem p g) = SubProblem p (fmap f . g)

instance Applicative Problems where
  pure = Done
  pf <*> px = pf >>= \f -> px >>= \x -> return (f x)

instance Monad Problems where
  return = Done
  Fail >>= g = Fail
  Done x >>= g = g x
  SubProblem p f >>= g = SubProblem p (\x -> f x >>= g)
```

This monad instance basically just codes up concatenation of problems. With lists of judgments, we can just concatenate them, but what to do with the functions that construct results? Here instead we say that if we have one sequence of problems that produces some result, and from that result, we can compute another sequence of problems, well we can just dig around in the first sequence and replace its Done node (which ends the sequence of problems with the result) by the problems we would get. We thus get a single big sequence of problems.

This monadic interfaces also gives us a really elegant way of writing these telescopes:

```
subProblem :: Judgment r -> Problems r
subProblem j = SubProblem j (\x -> Done x)

decomposePlus12 :: Nat -> Nat -> Problems Nat
decomposePlus12 Zero     y = return y
decomposePlus12 (Suc x) y =
  do z <- subProblem (Plus12 x y)
     return (Suc z)

decomposePlus13 :: Nat -> Nat -> Problems Nat
decomposePlus13 Zero z = return z
decomposePlus13 (Suc x) (Suc z) =
  subProblem (Plus13 x z)
decomposePlus13 _ _ = Fail
```

Let's add in a full ternary predicate version of our Plus to see how this works with other kinds of returned values:

```
data Judgment r where
  Plus12 :: Nat -> Nat -> Judgment Nat
  Plus13 :: Nat -> Nat -> Judgment Nat
  Plus123 :: Nat -> Nat -> Nat -> Judgment ()

decomposePlus123 :: Nat -> Nat -> Nat -> Problems ()
decomposePlus123 Zero y z =
  if y == z
     then return ()
     else Fail
decomposePlus123 (Suc x) y (Suc z) =
  subProblem (Plus123 x y z)
```

Readers who are especially familiar with functional programming idioms will observe that this is a variety of free monad construct, namely, the call-response tree variety.

And now, what parts of this really depend on the problem domain of addition? Well, clearly Judgment, because that defines what the problems are in the first place. And of course, as a result of that, the various decomposition functions. But not much else, provided we have some means of abstracting over those. Namely: the Problems type can be generalized away from Judgment, and findProof can be generalized away from the implementation of decompose, by way of a type class.

```
data Problems (j :: * -> *) (r :: *) where
  Fail :: Problems j r
  Done :: r -> Problems j r
  SubProblem :: j s -> (s -> Problems j r) -> Problems j r
```

```
subProblem :: j r -> Problems j r
subProblem j = SubProblem j (\x -> Done x)

class Decomposable j where
  decompose :: j r -> Problems j r

findProof :: Decomposable j => j r -> Maybe r
findProof j = solveProblems (decompose j)

solveProblems :: Decomposable j => Problems j r -> Maybe r
solveProblems Fail = Nothing
solveProblems (Done x) = Just x
solveProblems (SubProblem j f) =
  do x <- findProof j
     solveProblems (f x)
```

Having abstracted this far, we now can extract this into a little library and use this for bidirectional proof systems in general. Let's now tackle the simply typed lambda calculus.

Simply Typed LC

Because we've extracted out the proof refinement toolkit, we need to only give definitions for the judgments and decomposition of our lambda calculus. This is a great simplification from the setting before. We can now express that type checking is a judgment that produces no interesting information, but that type synthesis will give us some type information:

```
data Judgment r where
  Check :: [(String,Type)] -> Program -> Type -> Judgment ()
  Synth :: [(String,Type)] -> Program -> Judgment Type
```

Our decompositions are now more interesting as well, and hopefully a bit more insightful:

```
decomposeCheck :: [(String,Type)] -> Program -> Type -> Problems Judgment ()
decomposeCheck g (Pair m n) (Prod a b) =
  do subProblem (Check g m a)
     subProblem (Check g n b)
decomposeCheck g (Lam x m) (Arr a b) =
  subProblem (Check ((x,a):g) m b)
decomposeCheck g m a =
  do a2 <- subProblem (Synth g m)
     if a == a2
        then return ()
        else Fail


decomposeSynth :: [(String,Type)] -> Program -> Problems Judgment Type
decomposeSynth g (Var x) =
  case lookup x g of
    Nothing -> Fail
    Just a -> return a
decomposeSynth g (Ann m a) =
  do subProblem (Check g m a)
     return a
decomposeSynth g (Fst p) =
  do t <- subProblem (Synth g p)
     case t of
       Prod a b -> return a
       _ -> Fail
decomposeSynth g (Snd p) =
  do t <- subProblem (Synth g p)
     case t of
       Prod a b -> return b
       _ -> Fail
decomposeSynth g (App f x) =
  do t <- subProblem (Synth g f)
     case t of
       Arr a b ->
         do subProblem (Check g x a)
            return b
       _ -> Fail
decomposeSynth g m = Fail


instance Decomposable Judgment where
  decompose (Check g m a) = decomposeCheck g m a
  decompose (Synth g m) = decomposeSynth g m
```

And we're done! That is the full definition of the type checker for the simply typed lambda calculus with pairs and functions! It has the benefit of being fairly straightforward to read.

Conclusion

This wraps up the series of blog posts on proof refinement. One limitation to this approach is that errors are uninformative, but we can actually modify this toolkit to provide not just informative errors (**Either** instead of **Maybe**), but highly informative context-aware errors that know what subproblems are being worked on. Another limitation is that the above toolkit only works for when there is at most one result from the bottom-up direction. That is to say, either a judgment has no proofs, and so there's no bottom-up result, or it has exactly one proof and thus one bottom-up result. But we might have multiple such results, for instance, we might have instead built a system for addition that has the first two arguments of **Plus** as the bottom-up results (i.e. solutions for Plus3 c), and we'd like to be able to get out all pairs (`x,y`) such that `x + y = c` for fixed c. There are plenty of those pairs, so we had better be able to get some kind of list-like results. We also might imagine some other kind of system where in the course of constructing a proof we need to invent something out of thin air, such as a new name for a variable. In that kind of setting we'd like to have a proof system that could make use of some state for the collection of generated names. I'll look at both of these limitations in future blog posts.

If you have comments or questions, get it touch. I'm @psygnisfive on Twitter, augur on freenode (in #languagengine and #haskell).

This post is the third part of a three part series, the first post is Proof Refinement Basics, and the second post is Bidirectional Proof Refinement.