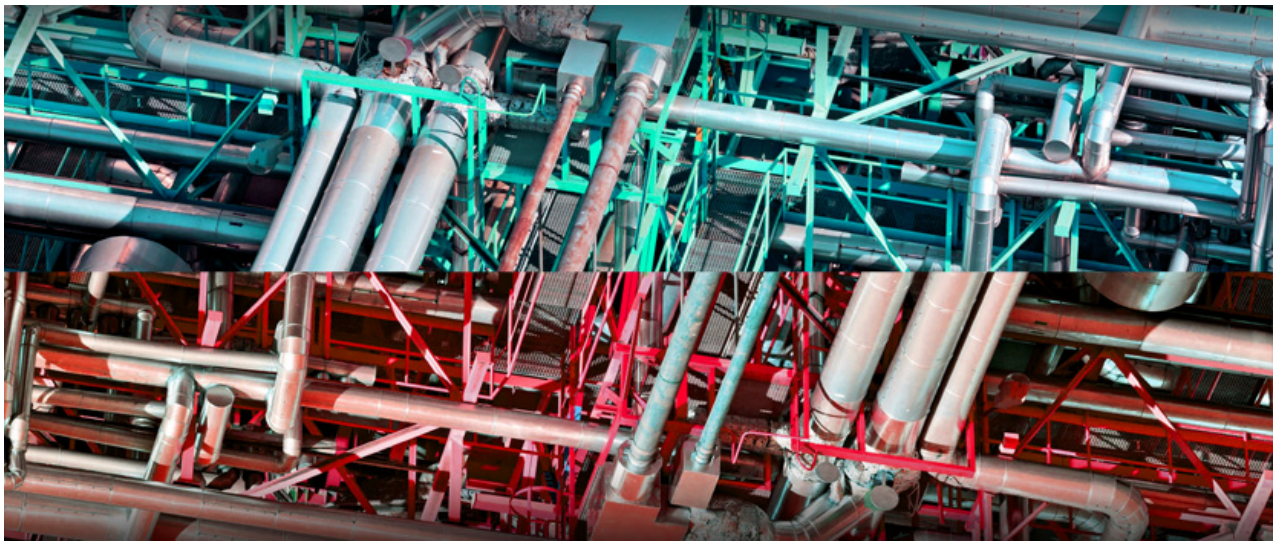
IOHK | CARDANO BLOG

Bidirectional proof refinement

MARCH 16, 2017 DARRYL MCADAMS 13 MIN READ 0 COMMENTS



In my last blog post, we looked at the very basics of proof refinement systems. I noted at the end that there was a big drawback of the systems as shown, namely that information flows only in one direction: from the root of a proof tree to the leaves. In this post, we'll see how to get information to flow in the opposite direction as well.

The code for this blog post can be found on GitHub here.

Addition Again

The addition judgment `Plus` which we defined last time was a three-argument judgment. The judgment `Plus L M N` was a claim that `N` is the sum of `L` and `M`. But we might want to instead think about specifying only some of the arguments to this judgment. In a language like Prolog, this is implemented by the use of metavariables which get valued in the course of computation. In the proof refinement setting, however, we take a different approach. Instead, we make use of a notion called bidirectionality, where we specify that a judgment's arguments can come in two modes: input and output. This is somewhat related to viewing thing

from a functional perspective, but in a more non-deterministic or relational fashion.

Let's start by deciding that the mode of the first two arguments to `Plus` is that of input, and the third is output. This way we'll think about actually computing the sum of two numbers. Instead of three `Nat` inputs, we'll instead use only two. We'll use a judgment name that reflects which two:

```haskell
-- Haskell

data Judgment = Plus12 Nat Nat
  deriving (Show)
```

```javascript
// JavaScript

function Plus12(x,y) {
    return { tag: "Plus12", args: [x,y] };
}
```

The behavior of the decomposer for `Plus12` will have to be slightly different now, though, because there are only two arguments. But additionally, we want to not only expand a goal into subgoals, but also somehow synthesize a result from the results of the subgoals. So where before we had only to map from a goal to some new goals, now we must also provide something that maps from some subresults to a result.

Rather than giving two separate functions, we'll give a single function that returns both the new subgoals, and the function that composes a value from the values synthesized for those subgoals.

```haskell
-- Haskell
decomposePlus12 :: Nat -> Nat -> Maybe ([Judgment], [Nat] -> Nat)
decomposePlus12 Zero y = Just ([], \zs -> y)
decomposePlus12 (Suc x) y = Just ([Plus12 x y], \[z] -> Suc z)

decompose :: Judgment -> Maybe ([Judgment], [Nat] -> Nat)
decompose (Plus12 x y) = decomposePlus12 x y
```

```javascript
// JavaScript

function decomposePlus12(x,y) {
    if (x.tag === "Zero") {
        return Just([[], zs => y]);
    } else if (x.tag === "Suc") {
        return Just([[Plus12(x.arg,y)], zs => Suc(zs[0])]);
    }
}

function decompose(j) {
    if (j.tag === "Plus12") {
        return decomposePlus12(j.args[0], j.args[1]);
    }
}
```

The `decompose` function is somewhat redundant in this example, but the shape of the problem is preserved by having this redundancy. The same is true of the `Maybe` in the types. We can always decompose now, so

the `Nothing` option is never used, but I'm keeping it here to maintain parallelism with the general framework.

Building a proof tree in this setting proceeds very similarly to how it did before, except now we need to make use of the synthesis functions in parallel with building a proof tree:

```haskell
-- Haskell

findProof :: Judgment -> Maybe (ProofTree, Nat)
findProof j =
  case decompose j of
    Nothing -> Nothing
    Just (js, f) -> case sequence (map findProof js) of
      Nothing -> Nothing
      Just tns -> let (ts, ns) = unzip tns
                  in Just (ProofTree j ts, f ns)
```

```javascript
// JavaScript

function unzip(xys) {
    var xs = [];
    var ys = [];

    for (var i = 0; i < xys.length; i++) {
        xs.push(xys[i][0]);
        ys.push(xys[i][1]);
    }

    return [xs,ys]
}

function findProof(j) {
    var mjs = decompose(j);

    if (mjs.tag === "Nothing") {
        return Nothing;
    } else if (mjs.tag === "Just") {
        var js = mjs.arg[0]
        var f = mjs.arg[1]
        var mtns = sequence(js.map(j => findProof(j)));

        if (mtns.tag === "Nothing") {
            return Nothing;
        } else if (mtns.tag === "Just") {
            var tsns = unzip(mtns.arg);
            return Just([ProofTree(j, tsns[0]), f(tsns[1])]);
        }
    }
}
```

Now when we run this on some inputs, we get back not only a proof tree, but also the resulting value, which is the `Nat` that would have been the third argument of `Plus` in the previous version of the system.

Let's now add another judgment, `Plus13`, which will synthesize the second argument of the original `Plus` judgment from the other two. Therefore, the judgment `Plus13 L N` means `L` subtracted from `N`. We'll add this judgment to the existing `Judgment` declarations.

```haskell
-- Haskell

data Judgment = Plus12 Nat Nat | Plus13 Nat Nat
```

```
  deriving (Show)
```

```
// JavaScript

function Plus13(x,z) {
    return { tag: "Plus13", args: [x,z] };
}
```

We can now define a decomposition function for this judgment. Notice that this one is partial, in that, for some inputs, there's no decomposition because the first argument is larger than the second. We'll also extend the `decompose` function appropriately.

```
-- Haskell

decomposePlus13 :: Nat -> Nat -> Maybe ([Judgment], [Nat] -> Nat)
decomposePlus13 Zero z = Just ([], \xs -> z)
decomposePlus13 (Suc x) (Suc z) = Just ([Plus13 x z], \[x] -> x)
decomposePlus13 _ _ = Nothing

decompose :: Judgment -> Maybe ([Judgment], [Nat] -> Nat)
decompose (Plus12 x y) = decomposePlus12 x y
decompose (Plus13 x z) = decomposePlus13 x z
```

```
// JavaScript

function decomposePlus13(x,z) {
    if (x.tag === "Zero") {
        return Just([[], xs => z]);
    } else if (x.tag === "Suc" && z.tag === "Suc") {
        return Just([[Plus13(x.arg, z.arg)], xs => xs[0]]);
    } else {
        return Nothing;
    }
}

function decompose(j) {
    if (j.tag === "Plus12") {
        return decomposePlus12(j.args[0], j.args[1]);
    } else if (j.tag === "Plus13") {
        return decomposePlus13(j.args[0], j.args[1]);
    }
}
```

If we now try to find a proof for `Plus13 (Suc Zero) (Suc (Suc (Suc Zero)))`, we get back `Suc (Suc Zero)`, as expected, because 1 subtracted from 3 is 2. Similarly, if we try to find a proof for `Plus13 (Suc (Suc Zero)) (Suc Zero)`, we find that we get no proof, because we can't subtract a number from something smaller than it.

Type Checking Again

We'll aim to do the same thing with the type checker as we did with addition. We'll split the `HasType` judgment into two judgments, `Check` and `Synth`. The `Check` judgment will be used precisely in those cases where a type must be provided for the proof to go through, while the `Synth` judgment will be used for cases where the structure of the program is enough to tell us its type.

```haskell
-- Haskell

data Judgment = Check [(String,Type)] Program Type
              | Synth [(String,Type)] Program
  deriving (Show)
```

```javascript
// JavaScript

function Check(g,m,a) {
    return { tag: "Check", args: [g,m,a] };
}

function Synth(g,m) {
    return { tag: "Synth", args: [g,m] };
}
```

Additionally, because of these changes in information flow, we'll remove some type annotations from the various program forms, and instead introduce a general type annotation form that will be used to shift between judgments explicitly.

```haskell
-- Haskell

data Program = Var String | Ann Program Type
             | Pair Program Program | Fst Program | Snd Program
             | Lam String Program | App Program Program
  deriving (Show)
```

```javascript
// JavaScript

function Var(x) {
    return { tag: "Var", arg: x };
}

function Ann(m,a) {
    return { tag: "Ann", args: [m,a] };
}

function Pair(m,n) {
    return { tag: "Pair", args: [m,n] };
}

function Fst(m) {
    return { tag: "Fst", arg: m };
}

function Snd(m) {
    return { tag: "Snd", arg: m };
}

function Lam(x,m) {
    return { tag: "Lam", args: [x,m] };
}

function App(m,n) {
    return { tag: "App", args: [m,n] };
}
```

The last change that we'll make will be to change the notion of synthesis a little bit from what we had before. In the addition section, we said merely that we needed a function that synthesized a new value from some subvalues. More generally, though, we need that process to be able to fail, because we wish to place

constraints on the synthesized values. To capture this, we'll wrap the return type in `Maybe`.

```haskell
-- Haskell

decomposeCheck
  :: [(String,Type)]
  -> Program
  -> Type
  -> Maybe ([Judgment], [Type] -> Maybe Type)
decomposeCheck g (Pair m n) (Prod a b) =
  Just ([Check g m a, Check g n b], \as -> Just undefined)
decomposeCheck g (Lam x m) (Arr a b) =
  Just ([Check ((x,a):g) m b], \as -> Just undefined)
decomposeCheck g m a =
  Just ( [Synth g m]
       , \[a2] -> if a == a2 then Just undefined else Nothing
       )

decomposeSynth
  :: [(String,Type)]
  -> Program
  -> Maybe ([Judgment], [Type] -> Maybe Type)
decomposeSynth g (Var x) =
  case lookup x g of
    Nothing -> Nothing
    Just a -> Just ([], \as -> Just a)
decomposeSynth g (Ann m a) =
  Just ([Check g m a], \as -> Just a)
decomposeSynth g (Fst p) =
  Just ( [Synth g p]
       , \[t] -> case t of
                   Prod a b -> Just a
                   _ -> Nothing
       )
decomposeSynth g (Snd p) =
  Just ( [Synth g p]
       , \[t] -> case t of
                   Prod a b -> Just b
                   _ -> Nothing
       )
decomposeSynth g (App f x) =
  Just ( [Synth g f, Synth g x]
       , \[s,t] -> case s of
                     Arr a b | a == t -> Just b
                     _ -> Nothing)
decomposeSynth _ _ = Nothing

decompose :: Judgment -> Maybe ([Judgment], [Type] -> Maybe Type)
decompose (Check g m a) = decomposeCheck g m a
decompose (Synth g m) = decomposeSynth g m
```

```javascript
// JavaScript

function eq(x,y) {
  if (x instanceof Array && y instanceof Array) {
    if (x.length != y.length) { return false; }

    for (var i = 0; i < x.length; i++) {
      if (!eq(x[i], y[i])) { return false; }
    }

    return true;
  } else if (x.tag === y.tag) {
    if (x.args && y.args) {
      return eq(x.args,y.args);
    } else if (x.arg && y.arg) {
      return eq(x.arg,y.arg);
    } else if (!x.arg && !y.arg) {
      return true;
    } else {
      return false;
    }
  }
```

```
}

function decomposeCheck(g,m,a) {
    if (m.tag === "Pair" && a.tag === "Prod") {
        return Just([ [Check(g, m.args[0], a.args[0]),
                       Check(g, m.args[1], a.args[1])],
                      as => Just(undefined)]);
    } else if (m.tag === "Lam" && a.tag === "Arr") {
        return Just([ [Check([[m.args[0], a.args[0]]].concat(g),
                             m.args[1],
                             a.args[1])],
                      as => Just(undefined)]);
    } else {
        return Just([ [Synth(g,m,a)],
                      as => eq(a,as[0]) ? Just(undefined) : Nothing ])
    }
}

function decomposeSynth(g,m) {
    if (m.tag === "Var") {
        var ma = lookup(m.arg, g);

        if (ma.tag === "Nothing") {
            return Nothing;
        } else if (ma.tag === "Just") {
            return Just([[], as => Just(ma.arg)]);
        }
    } else if (m.tag === "Ann") {
        return Just([ [Check(g, m.args[0], m.args[1])],
                      as => Just(m.args[1]) ]);
    } else if (m.tag === "Fst") {
        return Just([ [Synth(g, m.arg)],
                      as => as[0].tag === "Prod" ?
                              Just(as[0].args[0]) :
                              Nothing ]);
    } else if (m.tag === "Snd") {
        return Just([ [Synth(g, m.arg)],
                      as => as[0].tag === "Prod" ?
                              Just(as[0].args[1]) :
                              Nothing ]);
    } else if (m.tag === "App") {
        return Just([ [Synth(g, m.args[0]), Synth(g, m.args[1])],
                      as => as[0].tag === "Arr" &&
                              eq(as[0].args[0], as[1]) ?
                                Just(as[0].args[1]) :
                                Nothing ]);
    } else {
        return Nothing;
    }
}

function decompose(j) {
    if (j.tag === "Check") {
        return decomposeCheck(j.args[0], j.args[1], j.args[2]);
    } else if (j.tag === "Synth") {
        return decomposeSynth(j.args[0], j.args[1]);
    }
}
```

Finally, the `findProof` function has to be augmented slightly to deal with the new `Maybe` in the return type of the synthesizing function:

```
-- Haskell

findProof :: Judgment -> Maybe (ProofTree, Type)
findProof j =
  case decompose j of
    Nothing -> Nothing
    Just (js,f) -> case sequence (map findProof js) of
      Nothing -> Nothing
      Just tsas ->
        let (ts,as) = unzip tsas
        in case f as of
             Nothing -> Nothing
             Just a -> Just (ProofTree j ts, a)
```

```
// JavaScript

function findProof(j) {
    var mjs = decompose(j);

    if (mjs.tag === "Nothing") {
        return Nothing;
    } else if (mjs.tag === "Just") {
        var js = mjs.arg[0]
        var f = mjs.arg[1]
        var mtns = sequence(js.map(j => findProof(j)));

        if (mtns.tag === "Nothing") {
            return Nothing;
        } else if (mtns.tag === "Just") {
            var tsns = unzip(mtns.arg);
            var mn = f(tsns[1]);

            if (mn.tag === "Nothing") {
                return Nothing;
            } else if (mn.tag === "Just") {
                return Just([ProofTree(j, tsns[0]), mn.arg]);
            }
        }
    }
}
```

If we now try to find proofs for some typical synthesis and checking examples, we find what we expect:

```
findProof (Check [] (Lam "p" (Fst (Var "p"))) (Arr (Prod Nat Nat) Nat))

findProof (Check [] (Lam "p" (Fst (Var "p"))) (Arr Nat Nat))

findProof (Synth [("p",Prod Nat Nat)] (Fst (Var "p")))
```

Conclusion

The bidirectional style in this post has some interesting limitations and drawbacks. Consider the case of the rule for `App`: it has to synthesize the type of both the function and the argument and then compare them appropriately. This means that certain programs require additional type annotations to be written. But this is somewhat unnecessary, because once we synthesize the type of the function, we can use that information to *check* the argument. In the next blog post, I'll try to explain how to do this in a more elegant way, the combines both the upward and downward flows of information.

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

March 16, 2017 by darryl.mcadams@iohk.io

Bidirectional proof refinement

March 07, 2017 by darryl.mcadams@iohk.io

Proof refinement basics