IOHK | BLOG

# Proof Refinement Basics

MARCH 01, 2017 13 MIN READ

In this blog post, I'm going to discuss the overall structure of a proof refinement system. Such systems can be used for implementing automatic theorem provers, proof assistants, and type checkers for programming languages. The proof refinement literature is either old or hard to understand, so this post, and subsequent ones on the same topic, will present it in a more casual and current way.

Work on proof refinement as a methodology for building type checkers and interactive proof systems has a long history, going back to LCF, HOL, Nuprl, and Coq. These techniques never really penetrated into the mainstream of programming language implementation, but perhaps this may change.

The GitHub repo for this project can be found here.

## Prologue

As part of my work for IOHK, I've been designing and implementing a programming language called Plutus, which we use as the scripting language for our blockchains. Because IOHK cares deeply about the correctness of its systems, Plutus needs to be held to a very high standard, and needs to enable easy reasoning about its behavior. For that reason, I chose to make Plutus a pure, functional language with a static type system. Importantly, the language has a formal type theoretic specification.

To implement the type checker for the language, I used a fairly standard technique. However, recently I built a new framework for implementing programming languages which, while different from the usual way, has a more direct connection to the formal specification. This greater similarity makes it easier to check that the implementation is correct.

The framework also makes a number of bugs easier to eliminate. One class of bugs that arose a number of times in the first implementation was that of metavariables and unification. Because the language supports a certain about of unification-driven type inference, it's necessary to propagate updates to the information state of the type checking algorithm, so that everything is maximally informative and the algorithm can run correctly. Propagating this information by hand is error prone, and the new framework makes it possible to do it once and for all, in a bug-free way.

The framework additionally makes some features easier to program. For example, good error reporting is extremely important. But good error messages need to have certain information about where the error occurs. Not just where in the source code, but where in the logical structure of the program. All sorts of important information about the nature of the error, and the possible solutions, depend on that. The framework I developed also makes this incredibly easy to implement, so much so that it's build right into the system, and any language implemented using it can take advantage of it.

# Proofs and Proof Systems

In the context of modern proof theory and type theory, a proof can be seen as a tree structure with labeled nodes and the class of proof trees which are valid is defined by a set of rules that specify how a node in a tree may relate to its child nodes. The labels, in the context of proofs, are usually hypothetical judgments, and the rules are inference rules, but you can also use the same conceptual framework for many other things.

An example of a rule is **`for any choice of A and B, if you can show that A is true and that B is true, then it's permissible to conclude A∧B is true`**. This is usually written with the notation

$$\frac{A\ \text{true} \qquad B\ \text{true}}{A \wedge B\ \text{true}}\ \wedge\text{I}$$

This rule explains one way that it's ok to have a node labeled A∧B is true in a proof tree, namely, when its child nodes are **`A is true`** and **`B is true`**. So this rule justifies the circled node in the following tree:

$$\dfrac{\dfrac{\dfrac{}{B \wedge A\ \text{true}}\,\text{hyp(p)}}{A\ \text{true}}\,\wedge\text{E2} \qquad \dfrac{\dfrac{}{B \wedge A\ \text{true}}\,\text{hyp(p)}}{B\ \text{true}}\,\wedge\text{E1}}{A \wedge B\ \text{true}}\,\wedge\text{I}$$
$$\frac{A \wedge B\ \text{true}}{(B \wedge A) \supset (A \wedge B)\ \text{true}}\,\supset\text{I(p)}$$

When used as a tool for *constructing* a proof, however, we think about rule use in a directional fashion. We start from the conclusion — the parent node — because this is what we'd like to show, and we work backwards to the premises of the rule — the child nodes — to justify the conclusion. We think of this as a dynamic process of *building* a proof tree, not simply *having* one. From that perspective, the rule says that to build a proof tree with the root labeled **`A&B is true`**, we can expand that node to have two child nodes **`A is true`** and **`B is true`**, and then try to build those trees.

In the dynamic setting, then, this rule means that we may turn the first tree below into the second:

$$\frac{A \wedge B \text{ true}}{(B \wedge A) \supset (A \wedge B) \text{ true}} \supset I(p)$$

$$\frac{\dfrac{A \text{ true} \qquad B \text{ true}}{A \wedge B \text{ true}} \wedge I}{(B \wedge A) \supset (A \wedge B) \text{ true}} \supset I(p)$$

In this setting, though, It's important to be aware of a distinction between two kinds of nodes: nodes which may appear in finished proof trees as is, and nodes which must still expanded before they can appear in a finished proof tree. In the above trees, we indicated this by the fact that the nodes which may appear in finished proof trees have a bar above them labeled by the name of the rule that justified them. Nodes which must still be expanded had no such bar, and we call them *goals*.

# Building a Proof Refinement System

Let's now build a proof refinement system for a toy system, namely the system of equality (`x == y`) and addition (`x + y = z`) as relations among natural numbers. We'll implement it in both Haskell and JavaScript to give a sense of how things work. We'll start by just making the proof refiner system, with no interactivity nor any automatic driving of the proof system. We'll have to rely on the REPL to manage these things for us.

So the first thing we need is a definition of the naturals. In Haskell, we'll just use a normal algebraic data type, but in JavaScript, we'll use an object with a mandatory `tag` key and an optional `arg` key, together with helpers:

```haskell
-- Haskell

data Nat = Zero | Suc Nat
  deriving (Show)
```

```javascript
// JavaScript

var Zero = { tag: "Zero" };

function Suc(n) {
    return { tag: "Suc", arg: n };
}
```

Similarly, we'll define the judgments in question:

```haskell
-- Haskell

data Judgment = Equal Nat Nat | Plus Nat Nat Nat
  deriving (Show)
```

```javascript
// JavaScript

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

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

We'll next need some way to decompose a statement such as `Plus (Suc Zero) (Suc Zero) Zero` into subproblems that would justify it. We'll do this by defining a function that performs the decomposition, returning a list of new goals. But, because the decomposition might fail, we'll wrap this in a `Maybe`.

```haskell
-- Haskell

decomposeEqual :: Nat -> Nat -> Maybe [Judgment]
decomposeEqual Zero Zero = Just []
decomposeEqual (Suc x) (Suc y) = Just [Equal x y]
decomposeEqual _ _ = Nothing

decomposePlus :: Nat -> Nat -> Nat -> Maybe [Judgment]
decomposePlus Zero y z = Just [Equal y z]
decomposePlus (Suc x) y (Suc z) = Just [Plus x y z]
decomposePlus _ _ _ = Nothing
```

```javascript
// JavaScript

var Nothing = { tag: "Nothing" };

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

function decomposeEqual(x,y) {
    if (x.tag === "Zero" && y.tag == "Zero") {
        return Just([]);
    } else if (x.tag === "Suc" && y.tag === "Suc") {
        return Just([Equal(x.arg, y.arg)]);
    } else {
        return Nothing;
    }
}

function decomposePlus(x,y,z) {
    if (x.tag === "Zero") {
      return Just([Equal(y,z)]);
    } else if (x.tag === "Suc" && z.tag === "Suc") {
        return Just([Plus(x.arg, y, z.arg)]);
    } else {
        return Nothing;
    }
}
```

If we explore what these mean when used, we can build some intuitions. If we want to decompose a judgment of the form `Plus (Suc Zero) (Suc Zero) (Suc (Suc (Suc Zero)))` in Haskell/`Plus(Suc(Zero), Suc(Zero), Suc(Suc(Suc(Zero))))` in JavaScript, we simply apply the `decomposePlus` function to the three arguments of the judgment:

```haskell
-- Haskell

decomposePlus (Suc Zero) (Suc Zero) (Suc (Suc (Suc Zero)))
  = Just [Plus Zero (Suc Zero) (Suc (Suc Zero))]
```

```javascript
// JavaScript

decomposePlus(Suc(Zero), Suc(Zero), Suc(Suc(Suc(Zero))))
  = Just([Plus(Zero, Suc(Zero), Suc(Suc(Zero)))])
```

We'll define a general decompose function that will inspect a judgment and then call the appropriate decomposition function:

```haskell
-- Haskell

decompose :: Judgment -> Maybe [Judgment]
decompose (Equal x y) = decomposeEqual x y
decompose (Plus x y z) = decomposePlus x y z
```

```javascript
// JavaScript

function decompose(j) {
    if (j.tag === "Equal") {
        return decomposeEqual(j.args[0], j.args[1]);
    } else if (j.tag === "Plus") {
        return decomposePlus(j.args[0], j.args[1], j.args[2]);
    }
}
```

Now let's define what constitutes a proof tree:

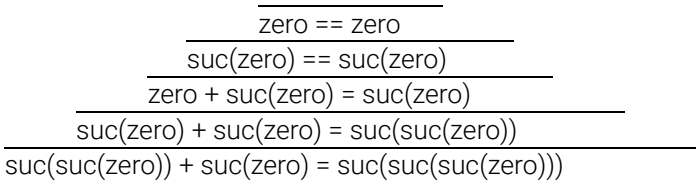
```haskell
-- Haskell

data ProofTree = ProofTree Judgment [ProofTree]
  deriving (Show)
```

```javascript
// JavaScript

function ProofTree(c,ps) {
    return { tag: "ProofTree", args: [c,ps] };
}
```

The first argument of the constructor **`ProofTree`** is the label for the node, which is the conclusion of inference that justifies the node. The second argument is the sub-proofs that prove each of the premises of inference. So for example, the proof tree




would be represented by

```haskell
-- Haskell

ProofTree (Plus (Suc (Suc Zero)) (Suc Zero) (Suc (Suc (Suc Zero))))
  [ ProofTree (Plus (Suc Zero) (Suc Zero) (Suc (Suc Zero)))
      [ ProofTree (Plus Zero (Suc Zero) (Suc Zero))
          [ ProofTree (Equal (Suc Zero) (Suc Zero))
              [ ProofTree (Equal Zero Zero)
                  []
              ]
          ]
      ]
  ]
```

```javascript
// JavaScript

ProofTree(Plus(Suc(Suc(Zero)), Suc(Zero), Suc(Suc(Suc(Zero)))),
  [ ProofTree(Plus(Suc(Zero), Suc(Zero), Suc(Suc(Zero))),
      [ ProofTree(Plus(Zero, Suc(Zero), Suc(Zero)),
          [ ProofTree(Equal(Suc(Zero), Suc(Zero)),
              [ ProofTree(Equal(Zero,Zero)
                  [])
              ])
          ])
      ])
  ])
```

We now can build a function, **`findProof`**, which will use **`decompose`** to try to find a proof for any judgment that it's given:

```haskell
-- Haskell

findProof :: Judgment -> Maybe ProofTree
findProof j =
  case decompose j of
    Nothing -> Nothing
    Just js -> case sequence (map findProof js) of
      Nothing -> Nothing
      Just ts -> Just (ProofTree j ts)
```

```javascript
// JavaScript

function sequence(mxs) {
    var xs = [];

    for (var i = 0; i < mxs.length; i++) {
        if (mxs[i].tag === "Nothing") {
            return Nothing;
        } else if (mxs[i].tag === "Just") {
            xs.push(mxs[i].arg);
```

```
        }
    }

    return Just(xs);
}

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

    if (mjs.tag === "Nothing") {
        return Nothing;
    } else if (mjs.tag === "Just") {
        var mts = sequence(mjs.arg.map(j => findProof(j)));
        if (mts.tag === "Nothing") {
            return Nothing;
        } else if (mts.tag === "Just") {
            return Just(ProofTree(j, mts.arg));
        }
    }
}
```

If we now run this on some example triples of numbers, we find that it returns `Just`s of proof trees exactly when the statement is true, and the trees show just how the statement is true. Try it on the above example for `suc(suc(zero)) + suc(zero) = suc(suc(suc(zero)))`.

The choice above to use the `Maybe` type operator reflects the idea that there is at most one proof of a judgment, but possible also none. If there can be many, then we need to use `List` instead, which would give us alternative definitions of the various functions:

```
-- Haskell

decomposeEqual :: Nat -> Nat -> [[Judgment]]
decomposeEqual Zero Zero = [[]]
decomposeEqual (Suc x) (Suc y) = [[Equal x y]]
decomposeEqual _ _ = []

decomposePlus :: Nat -> Nat -> Nat -> [[Judgment]]
decomposePlus Zero y z = [[Equal y z]]
decomposePlus (Suc x) y (Suc z) = [[Plus x y z]]
decomposePlus _ _ _ = []

decompose :: Judgment -> [[Judgment]]
decompose (Equal x y) = decomposeEqual x y
decompose (Plus x y z) = decomposePlus x y z

findProof :: Judgment -> [ProofTree]
findProof j =
  do js <- decompose j
     ts <- sequence (map findProof js)
     return (ProofTree j ts)
```

```
// JavaScript

function bind(xs,f) {
    if (xs.length === 0) {
        return [];
    } else {
        return f(xs[0]).concat(bind(xs.slice(1), f))
    }
}

function sequence(xss) {
    if (xss.length === 0) {
        return [[]];
    } else {
        return bind(xss[0], x =>
                 bind(sequence(xss.slice(1)), xs2 =>
                   [[x].concat(xs2)]));
```

```
    }
}

function findProof(j) {
    return bind(decompose(j), js =>
             bind(sequence(js.map(j2 => findProof(j2))), ts =>
               [ProofTree(j,ts)]));
}
```

In fact, Haskell lets us abstract over the choice of `Maybe` vs `List`, provided that the type operator is a monad.

# A Type Checker with Proof Refinement

Let's now move on to building a type checker. We'll first define the types we're going to use. We'll have natural numbers, product types (pairs), and arrow types (functions). We'll also use a BNF definition as a reference:

```
<type> ::= "Nat" | <type> "×" <type> | <type> "→" <type>
```

```
-- Haskell

data Type = Nat | Prod Type Type | Arr Type Type
  deriving (Show)
```

```
// JavaScript

var Nat = { tag: "Nat" };

function Prod(a,b) {
    return { tag: "Prod", args: [a,b] };
}

function Arr(a,b) {
    return { tag: "Arr", args: [a,b] };
}
```

Next we must specify what qualifies as a program, which we'll be type checking:

```
<program> ::= <name> | " ⟨" <program> "," <program> "⟩ "
            | "fst" <program> | "snd" <program>
            | "λ" <name> "." <program> | <program> <program>
```

```
-- Haskell

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

```
// JavaScript
```

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

var Zero = { tag: "Zero" };

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

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

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

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

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

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

The programs built with **`Fst`**, **`Snd`**, and **`App`** have more arguments that usual because we want to do only type checking, so we need to supply information that's not present in the types when working in a bottom up fashion. We'll see how we can avoid doing this later.

We'll also make use of a judgment **`HasType G M A`**, where **`G`** is a type context that assigns **`Type`**s to variables (represented by a list of string-type pairs), **`M`** is some program, and **`A`** is some type.

```
-- Haskell

data Judgment = HasType [(String,Type)] Program Type
  deriving (Show)
```

```
// JavaScript

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

We can now define the decompose functions for these. We'll just use the **`Maybe`** type for the decomposers here, because we don't need non-deterministic solutions. Type checkers either succeed or fail, and that's all.

```
-- Haskell

decomposeHasType
  :: [(String,Type)] -> Program -> Type -> Maybe [Judgment]
decomposeHasType g (Var x) a =
  case lookup x g of
    Nothing -> Nothing
    Just a2 -> if a == a2
               then Just []
               else Nothing
decomposeHasType g Zero Nat =
```

```
  Just []
decomposeHasType g (Suc m) Nat =
  Just [HasType g m Nat]
decomposeHasType g (Pair m n) (Prod a b) =
  Just [HasType g m a, HasType g n b]
decomposeHasType g (Fst b p) a =
  Just [HasType g p (Prod a b)]
decomposeHasType g (Snd a p) b =
  Just [HasType g p (Prod a b)]
decomposeHasType g (Lam x m) (Arr a b) =
  Just [HasType ((x,a):g) m b]
decomposeHasType g (App a m n) b =
  Just [HasType g m (Arr a b), HasType g n a]
decomposeHasType _ _ _ =
  Nothing

decompose :: Judgment -> Maybe [Judgment]
decompose (HasType g m a) = decomposeHasType g m a
```

```
// JavaScript

function lookup(x,xys) {
    for (var i = 0; i < xys.length; i++) {
        if (xys[i][0] === x) {
            return Just(xys[i][1]);
        }
    }

    return Nothing;
}

function decomposeHasType(g,m,a) {
    if (m.tag === "Var") {
        var ma2 = lookup(m.arg, g);

        if (ma2.tag === "Nothing") {
            return Nothing;
        } else if (ma2.tag === "Just") {
            return eq(a,ma2.arg) ? Just([]) : Nothing;
        }
    } else if (m.tag === "Zero" && a.tag === "Nat") {
      return Just([]);
    } else if (m.tag === "Suc" && a.tag === "Nat") {
      return Just([HasType(g, m.arg, Nat)]);
    } else if (m.tag === "Pair" && a.tag === "Prod") {
        return Just([HasType(g, m.args[0], a.args[0]),
                     HasType(g, m.args[1], a.args[1])])
    } else if (m.tag === "Fst") {
        return Just([HasType(g, m.args[1], Prod(a,m.args[0]))]);
    } else if (m.tag === "Snd") {
        return Just([HasType(g, m.args[1], Prod(m.args[0],a))]);
    } else if (m.tag === "Lam" && a.tag === "Arr") {
        return Just([HasType([[m.args[0],a.args[0]]].concat(g),
                             m.args[1],
                             a.args[1])]);
    } else if (m.tag === "App") {
        return Just([HasType(g, m.args[1], Arr(m.args[0],a)),
                     HasType(g, m.args[2], m.args[0])]);
    } else {
        return Nothing;
    }
}

function decompose(j) {
    if (j.tag === "HasType") {
        return decomposeHasType(j.args[0], j.args[1], j.args[2]);
    }
}
```

Using the same definitions of proof trees, sequencing, and finding a proof that we had earlier for **`Maybe`**, we can now perform some type checks on some programs. So for example, we can find that the program **`λx.x`** does indeed have the type **`Nat → Nat`**, or that it does not have the type **`(Nat × Nat) → Nat`**.

# Conclusion

That about wraps it up for the basics of proof refinement. One thing to observe about the above system is that its unidirectional: information flows from the bottom to the top, justifying expansions of goals into subtrees. No information flows back down from subtrees to parent nodes. In my next blog post, I'll discuss how we might go about adding bidirectional information flow to such a system. This will allow us to perform type checking computations where a judgment might return some information to be used for determining if a node expansion is justified. The practical consequence is that our programs become less cluttered with extraneous information.

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