Type Checking

1. Bidirectional Type Theories
   - 1.1 New Judgments
   - 1.2 Mode Checking
   - 1.3 Moded Pairs
     - 1.3.1 Mode Analysis
     - 1.3.2 Splitting $M : A$ into $A \ni M$ and $M \in A$
   - 1.4 Functions
   - 1.5 Ergonomics
2. Implementations

1. Bidirectional Type Theories

1.1 New Judgments

Generally speaking, type theories of the usual sort are not well suited to implementation. They're a good starting point to be sure! But they often have limitations. A common problem is that information needs to flow in arbitrary directions, typically relying on unification to achieve this.

The main alternative is to introduce two new judgments in place of the usual typing judgment $M : A$ as follows:

$\boxed{A \ni M}$ The type A checks term M. Both A and M are "inputs".

$\boxed{M \in A}$ The term M synthesizes the type A. M is an "input" while A is an "output".

Note: $A \ni M$ is sometimes written A chk M, M chk A, $M \leftarrow A$
$M \in A$ is sometimes written M syn A, $M \Rightarrow A$

This kind of system is called "bidirectional", because the logic is structured in such a way that different judgments can be seen as having definite input-output relationships amongst its metavars. Some metavars are seen as flowing "in" from below, and others as flowing "out" from above. This makes it easy to implement.

1.2 Mode Checking

The main tool we use to transform a non-directional proof system to a bidirectional one is called mode analysis. Given a particular judgment, e.g. plus m n p, we want to look at each metavar and ask if it can be "input" moded, that is, seen as an input, or "output" moded, seen as an output.

Let's pick a particular set of rules to work with for reference:

plus m n p Natural numbers m, n, and p add together such that $m + n = p$ is true.

$$\frac{}{\text{plus } 0\ n\ n.}\ \text{plus-0}$$

$$\frac{\text{plus } m\ n\ p}{\text{plus } (\text{suc } m)\ n\ (\text{suc } p)}\ \text{plus-suc}$$

Since the judgment has 3 metavariables, we could assign 8 different modes:

m, n, p are all inputs
m, n are inputs, p is output
m, p are inputs, n is output
etc.

1.2 Mode checking (cont.)

Each mode assignment would correspond to a different operational interpretation. For instance, if all metavars were input moded, this would correspond to a boolean test like "$m + n == p$", whereas if p was output moded this would correspond to the addition function. If only m were output moded, it would be the partial function '$\lambda p.\lambda n.\ p - n$'. If only p were input, it would be a set valued function that computes all the pairs (m,n) where $m+n=p$. Obviously the mode assignment has a major impact on the specific operational interpretation, despite them all being intimately related.

Let's use the mode assignment m,n input, p output to explain how to check if an assignment is possible/valid.

We first will look at the rule plus-0. There are no premises, so we ask:

> If we know all of the input moded metavars (m and n), does that fully determine the output moded metavars?
>
> Here, we know $m := 0$, but n is still a metavar in the rule, so if we knew that, would we know p?
>
> Yes! The rule explicitly reuses n in the p position, $p := n$, so knowing n means we know p.

1.2 Mode Checking (cont.)

We now continue on to the rule plus-suc. This rule has premises, so something interesting happens:

(1) If we know all of the input moded meta vars in the conclusion, does that mean that the input metavars of the first premise are known? → Yes!
And the output metavars everywhere are unknown? → Yes!

(2) If we know the output metavar of the first premise, since it's also the last premise, is the output metavar of the conclusion also known? → Yes!

Pictorially, we can represent "we know X, so we know Y" by drawing an arrow from one to the other:

(1) looks like

$$\frac{\text{plus } m \; n \; p}{\text{plus } (\text{suc } m) \; n \; (\text{suc } p)}\ \text{plus-suc}$$

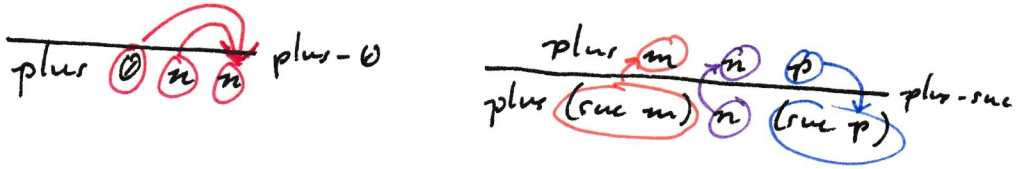
(2) looks like

$$\frac{\text{plus } m \; n \; p}{\text{plus } (\text{suc } m) \; n \; (\text{suc } p)}\ \text{plus-suc}$$

Knowledge never flows into a metavar position twice, never flows into a conclusion input position, nor into a premise output position. Knowledge also always flows into premise inputs, and out of premise outputs.

1.2 Mode Checking (cont.)

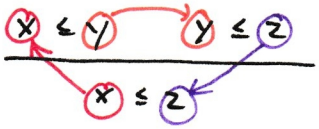
These little diagrams are really convenient so here's one for plus-0, and also a single unified diagram for plus-suc:



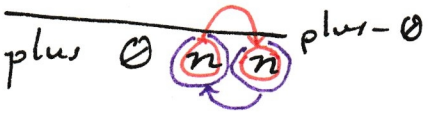
When a rule has multiple premises, knowledge can flow sideways to later judgments, from left outputs to right inputs. For example, if the judgment $\underline{x \le y}$ has mode assignment x input, y output, with a rule

$$\frac{x \le y \qquad y \le z}{x \le z}\ \text{trans}$$

Then the information flow depicted here is fine:



On the other hand, conrider the mode arrignment m output, n,p input for the judgment plus m n p. When we look at rule plus-Ø, we have information flowing <u>into</u> an input moded poritirn: n flows into p, and vice versa:



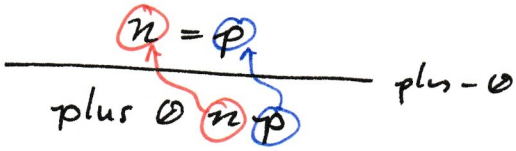
The judgment is certainly a perfectly sensible thing, so what can this mean? Where do n and p come from? The intention of this mode arrignment is to <u>ark</u> given n and p, what is m? So for this rule, we're basically saying that if n and p are the same, then m is Ø.

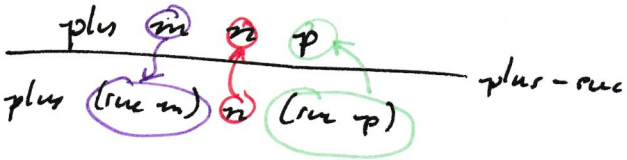
Operationally, however, we don't magically know two things are the same, we must check this. But this rule <u>lacks</u> an explizit check of equality, hence why the above mode analysis (in pictorial form) is problematic.

The solution is to add to plus-0 a new premise:

$$\frac{n = p}{\text{plus } 0\ n\ p}\ \text{plus-0 (modified)}$$

We can then say that the judgment $n = n$ has the mode assignment n, n input, and the mode assignment for plus $m\ n\ p$ will now be valid:





Returning to $x \leq y$, consider this rule:

$$\frac{}{0 \leq y}\ \text{base}$$

If x is input and y output as before, we don't <u>fully</u> know what y is just because we know x is 0.

Operationally, this is not obviously interpretable, but we can find an answer. Since y is unknown, the judgment holds, for all y, so we could understand this as some kind of non-deterministic operation "pick a number, any number". Alternatively, we could use a monadic interpretation to give this a set-based interpretation, provided we can compute all of the numbers (e.g. with laziness or generators).

Using mode checking, we can take an unmoded set of rules, and derive a new possibly larger set of moded rules, where judgments split into multiple distinct but related mode judgments. The unmoded $plus\ m\ n\ p$ might be split into two: $plus_{iii}\ m\ n\ p$ where all metavars are input (hence the subscripts), and $plus_{iio}\ m\ n\ p$ where only m and n are inputs. Our new rules might even mix the different moded judgments, replacing some original unmoded judgments with one new moded judgment, and some with another.

1.3 Moded Pairs

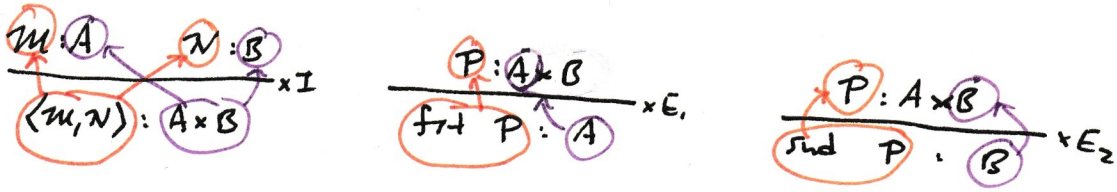
1.3.1 Mode Analysis

Let's now do a mode analysis of the following typing fragment:

$$\boxed{M : A}$$

$$\frac{M : A \qquad N : B}{\langle M, N \rangle : A \times B}\ \times I \qquad \frac{P : A \times B}{\mathsf{fst}\ P : A}\ \times E_1 \qquad \frac{P : A \times B}{\mathsf{snd}\ P : B}\ \times E_2$$

Let's first check the mode assignment M is input and A is also input:

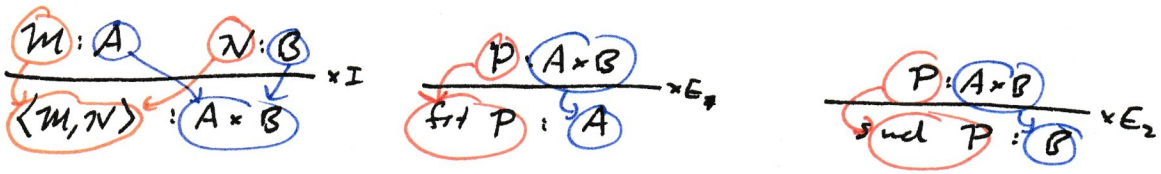


As we can see, in rule $\times E_1$, the B part of the premise is <u>not</u> known when we know the inputs of the conclusion. Similarly, for $\times E_2$, the A part is not known.

So this mode assignment is <u>fine</u> for $\times I$, but <u>bad</u> for $\times E_1$ and $\times E_2$!

1.3.1 Mode Analysis (cont.)

What about a different mode assignment, namely M is input, but A is output?



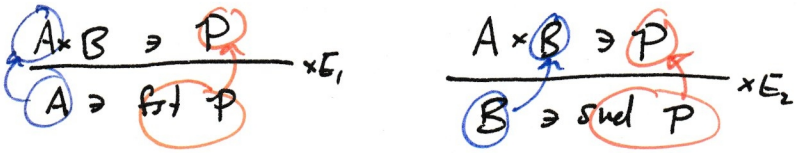
This mode assignment works just fine!

If we want a programming language with only pairs, this would be fine. But it will turn out that pairs are very special, and other types don't have this nice mode assignment.

1.3.2 Splitting $M : A$ into $A \ni M$ and $M \in A$

So with that in mind, let's see what it would look like to split the judgment $M : A$ into the two standard moded judgments $A \ni M$ and $M \in A$. M is input in both, A is output in $M \in A$.

We have to swap $M : A$ for $A \ni M$ and $M \in A$ in the rules, so have some choices to make. If we picked $A \ni M$ for both or either parts of $\times E_1$ and $\times E_2$, then we would run into the problems we saw previously, e.g.:



But choosing $M \in A$ for both conclusions and premises works just fine:

$$\frac{P \in A \times B}{\mathsf{fst}\ P \in A}\ \times E_1 \qquad \frac{P \in A \times B}{\mathsf{snd}\ P \in B}\ \times E_2$$

In general, it turns out that we should always pick $M \in A$ for elimination rules.

Since $\times I$ can be moded either way, let's pick $A \ni M$:

$$\frac{A \ni M \qquad B \ni N}{A \times B \ni \langle M, N \rangle}\ \times I$$

Introduction rules are generally best with $A \ni M$.

These mode assignments and modified rules are pretty good! But we've lost the ability to write some proofs. For instance, it used to be trivial to do this:

$$\frac{\dfrac{P : A \times B}{\mathsf{snd}\ P : B}\ \times E_2 \qquad \dfrac{P : A \times B}{\mathsf{fst}\ P : A}\ \times E_1}{\langle \mathsf{snd}\ P, \mathsf{fst}\ P \rangle : B \times A}$$

1.3.2 Splitting $M : A$ into $A \ni M$ and $M \in A$

But with these moded judgments and rules, we can't!

EXERCISE

Try to. Why does it fail?

To fix the problem, the standard move is to introduce one new syntactic form to the programs/terms, and two new rules:

$$\frac{M \in A' \quad A = A'}{A \ni M}\ \text{chk-syn} \qquad \frac{A \ni M}{(M : A) \in A}\ \text{syn-chk (aka annotation)}$$

↑ special syntax meant to evoke the judgment called a "type annotation"

We also need the judgment $A = B$, with both A and B as input moded. Since the definition/rules for $A = B$ are very straightforward, and the operational interpretation as an equality check is trivial, I'll omit further explanation of it here.

EXERCISE

Verify the mode assignments work with the rules chk-syn and syn-check

1.3.2 Splitting $M : A$ into $A \ni M$ and $M \in A$

We can now reconstruct a proof for the type of $\langle \mathrm{snd}\ P, \mathrm{fst}\ P \rangle$:

$$\dfrac{\dfrac{\mathrm{snd}\ P \in B \quad B = B'}{B \ni \mathrm{snd}\ P}\ \text{chk-syn} \qquad \dfrac{\mathrm{fst}\ P \in A' \quad A = A'}{A \ni \mathrm{fst}\ P}\ \text{chk-syn}}{B \times A \ni \langle \mathrm{snd}\ P, \mathrm{fst}\ P \rangle}\ \times I$$

Let's consider another proof that we could do before, and which needs the syn-chk rule to do now :

$$\dfrac{\dfrac{M : A \qquad N : B}{\langle M, N \rangle : A \times B}\ \times I}{\mathrm{fst}\ \langle M, N \rangle : A}\ \times E_1$$

This becomes :

$$\dfrac{\dfrac{\dfrac{A \ni M \qquad B \ni N}{A \times B \ni \langle M, N \rangle}\ \times I}{(\langle M, N \rangle : A \times B) \in A \times B}\ \text{syn-chk}}{\mathrm{fst}\ (\langle M, N \rangle : A \times B) \in A}\ \times E_1$$

1.3.2 Splitting $M : A$ into $A \ni M$ and $M \in A$

Hopefully you've noticed that the old proof is for a redex, and can be β-reduced. If not, notice so now.

This turns out to be a general pattern: we always need to use type annotations to create redexes in the new moded system, and we never need to use annotations if there are no redexes. Since it's quite rare for programs to have redexes already in them, type annotations have only a minor impact on programming ergonomics.

1.4 Functions

The usual definition for functions happens in a type theory with the hypothetical judgment $\Gamma \vdash M : A$ and looks like this:

$\boxed{\Gamma \vdash M : A}$ Term M has type A in context Γ

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.\, M : A \to B} \to I \qquad \frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash M\, N : B} \to E$$

These rules have trouble with mode assignments, however.

EXERCISE

Show that these mode assignments are invalid:

Γ, M, A are inputs

Γ, M are inputs, A is output

One very common solution is to put a type annotation on the formal parameter of the λ:

$$\frac{\Gamma, x{:}A \vdash M : B}{\Gamma \vdash \lambda x{:}A.M : A \to B} \to I \text{ (annotated)}$$

This choice makes the mode assignment Γ, M input, A output valid for all rules. However, this annotation will often be unnecessary from the perspective of the programmer. Consider the following function:

$$\lambda x{:}\mathrm{Int}.\ x+1$$

This has the type $\mathrm{Int} \to \mathrm{Int}$. The argument x can only be an Int since it's the first argument of $+$ which is an Int operation. The annotation should therefore be superfluous! It's <u>obvious</u> that x is an Int! But the rule above forces the annotation.

Unsurprisingly, a bidirectional system with $A \ni M$ and $M \in A$ lets us use the old unannotated λ's, and gives us a generic annotation mechanism for the rare case where it's needed:

1.4 Functions (cont.)

$$\frac{\Gamma, x{:}A \vdash B \ni M}{\Gamma \vdash A \to B \ni \lambda x.M} \to I \qquad \frac{\Gamma \vdash M \in A \to B \quad \Gamma \vdash A \ni N}{\Gamma \vdash M\,N \in B} \to E$$

The choice to have $A \ni N$ be the second premise of $\to E$, instead of $N \in A'$ and $A = A'$ really just comes down to ergonomics. Since the syn-chk rule lets us go from synthesis to checking as needed we get a more convenient system with checking, requiring fewer annotations.

EXERCISE

What modifications can be made to the original typing rules for functions to make the mode assignment Γ, M, A input valid?

1.5 Ergonomics

In general, languages can be designed to improve ergonomics by including "duplicate" typing rules to add more valid judgements. For example, we saw that $\times I$ could have both the mode assignments we looked at for the $M{:}A$ judgment.

In the bidirectional system, we only chose to make the rule use the moded judgment $A \ni M$, but the judgment $M \Rightarrow A$ would work just as well, like we saw. It would certainly make our lives a little easier if we could also synthesize the types of pairs, atleast sometimes, so why not have both sets of rules?

$$\frac{A \ni M \qquad B \ni N}{A \times B \ni \langle M, N \rangle} \times I \text{ (chk)}$$

$$\frac{P \in A \times B}{\text{fst } P \in A} \times E_1$$

$$\frac{M \in A \qquad N \in B}{\langle M, N \rangle \in A \times B} \times I \text{ (syn)}$$

$$\frac{P \in A \times B}{\text{snd } P \in B} \times E_2$$

The elims can only have synthesizing variants, however, so they don't change.

We can play similar games with any set of rules, adding in extraneous, but valid, rules to make our lives easier.

2. Implementations

We now move on to implementations. The aim is to show how to take a bidirectional system and translate it into an actual program. We'll start with the plus m n p judgment as a reference case.

plus m n p The sum of m and n is p.
Modes: m, n input
p output

$$\frac{}{\text{plus } 0\ n\ n}\ \text{plus-0}$$

$$\frac{\text{plus } m\ n\ p}{\text{plus } (\text{suc } m)\ n\ (\text{suc } p)}\ \text{plus-suc}$$

The translation proceeds as follows:

1. The judgment becomes a function where the input moded metavariables correspond to arguments and output moded metavariables to (tuples of) returned values:

```
plus :: Nat → Nat → Nat
```

- `plus` ← judgment name
- first `Nat` ← m's type, input mode, so arg
- second `Nat` ← n's type, input mod, so arg
- last `Nat` ← p's type, output mode, so returned

2. The rules become clauses in the definition of the function:

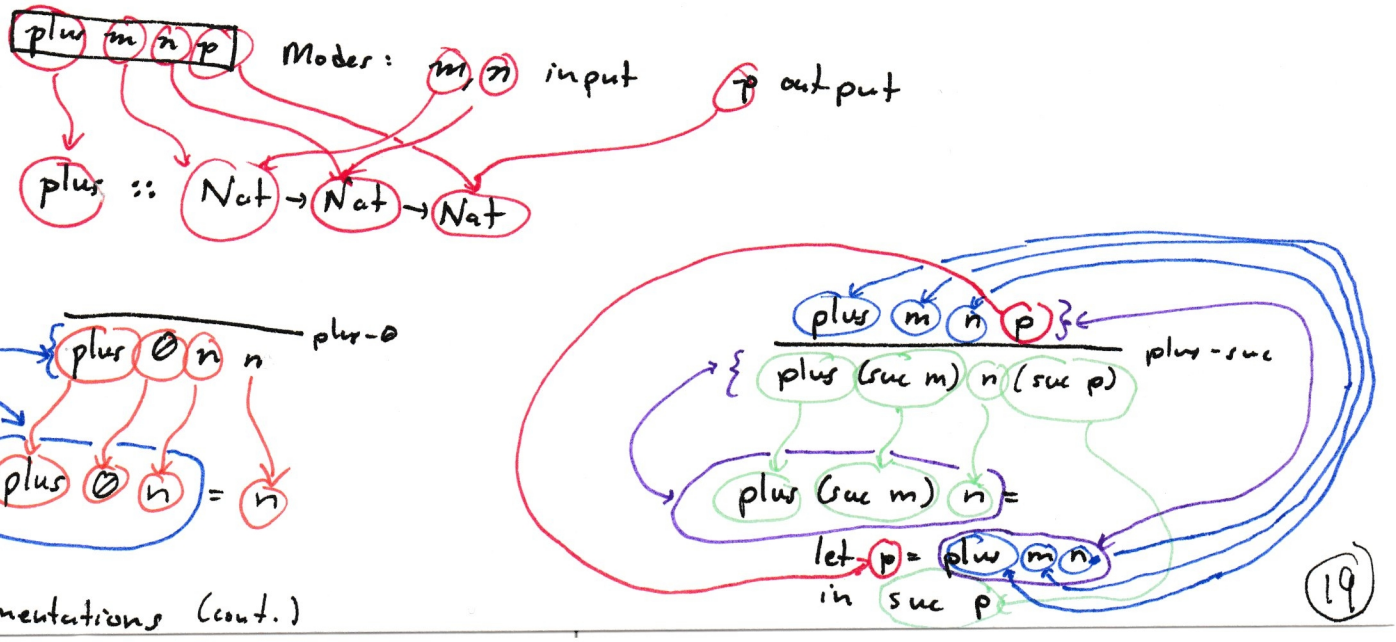
2.1 The conclusion becomes the left hand side (LHS) of the clause. Remember to omit the output moded metavars!

2.2 The premises become function calls in the right hand side (RHS) of the clause. Generally, their output moded metavars translate into bound results of function calls.

plus ⊘ n = n

plus (suc m) n = let p = plus m n in suc p

Schematically:



In the case of plus m n p, there's no peculiar operational properties: the inference rules' conclusions don't overlap, there are no rules with outputs coming from thin air, etc. and so the function can be a simple function. We can even simplify it to remove the let binding if we're ok with slightly obscuring the connection to the inference rules.

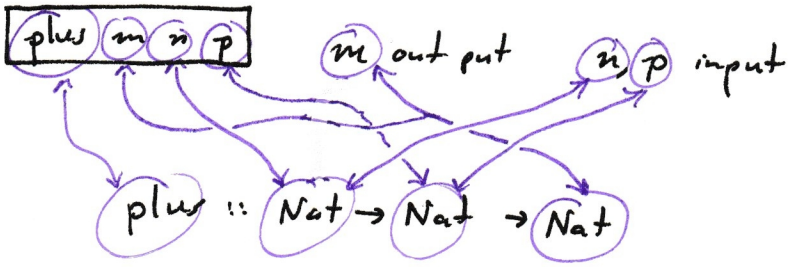
But some rules are a little peculiar. For instance, consider the version given below for a different mode assignment:

plus m n p The sum of m and n is p.
Modes: m output
n, p input

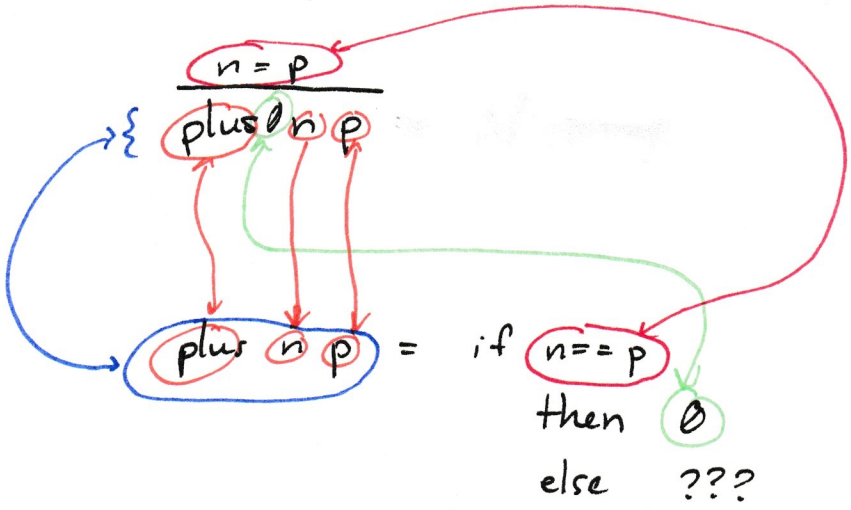
$$\frac{n = p}{\text{plus } \mathcal{O}\ n\ p}\ \text{plus-}\mathcal{O}$$

$$\frac{\text{plus } m\ n\ p}{\text{plus } (\text{suc } m)\ n\ (\text{suc } p)}\ \text{plus-suc}$$

Let's try to just directly apply the guidelines from before:



So far so good! What about rule plus-∅ ?



But how do we handle the case where $n \neq p$? The inference rule says that plus ∅ n p is provable only if $n = p$, and no other rule permits concluding plus ∅ n p for some n and p, therefore, when $n \neq p$, there is no proof at all. So what does this mean for the function? There should be no return value!

But that's often gnarly. The function would be partially defined. So instead, we can shift perspectives.

2. Implementations (cont.)

Instead of translating to a function that returns a Nat, we'll represent the possibility of unprovability with the Maybe type operator:

```
plus :: Nat → Nat → Maybe Nat
```

Our translation of plus-0 now is easy:

```
plus n p = if n == p
           then Just 0
           else Nothing
```

The translation of plus-suc can proceed similarly easilly. Because plus returns a maybe, we need to take its different values into consideration. We only get a natural number out if plus succeeds. But the premises of the plus-suc rule sort of already expects the sub-problem to be proven. So:

```
plus n (suc p) = case plus n p of
                   Nothing → Nothing
                   Just m  → Just (suc m)
```

But theres a problem: these two function equations overlap! For inference rules this is fine, because using a rule is a free choice to the prover, who is human. But the computer can't know the "right" choice.

2. Implementations (cont.)

In Haskell, the order of the equations imposes a choice, but how does that relate to what's provable? If we put them in one order, do we get different provable judgments than in the other order?

In this case, we can look at the intended meaning of the judgment:

plus m n p The sum of m and n is p.

We know that if $m+n=p$ then $n = p-m$, just from algebraic reasoning, so there can only ever be at most one value for n. But subtraction is only defined for some natural numbers.* In particular,

$$\begin{aligned}
&\emptyset - \emptyset = \emptyset \\
&\emptyset - \text{suc } m \quad \text{is undefined} \\
&\text{suc } p - \emptyset = \text{suc } p \\
&\text{suc } p - \text{suc } m = p - m \text{ if } p-m \text{ is defined} \\
&\qquad\qquad \text{and undefined otherwise}
\end{aligned}$$

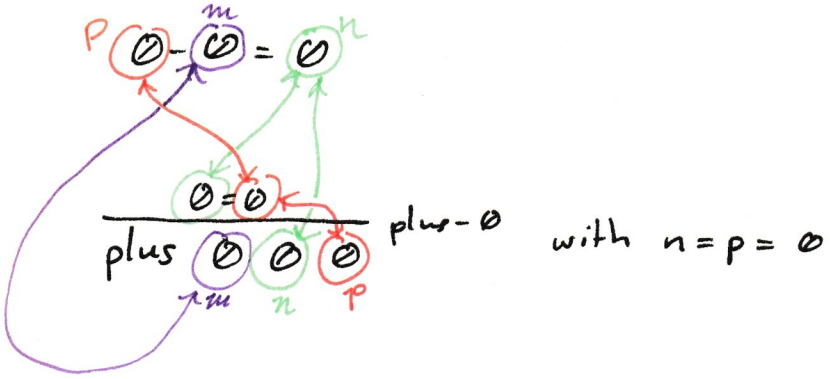
This accords with our definition of plus:

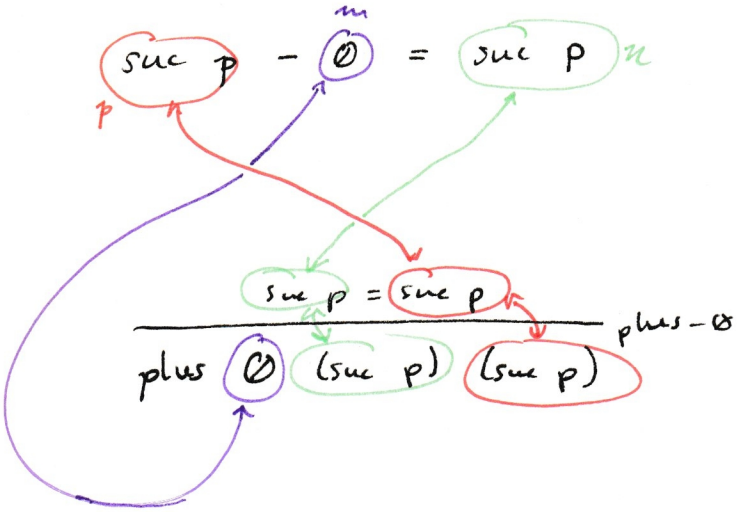
plus m n p is provable if and only if $m+n=p$, and therefore if plus m n p is provable, $n = p-m$ is defined.

---

*Some definitions of natural number subtraction set $\emptyset - m = \emptyset$ for all m.

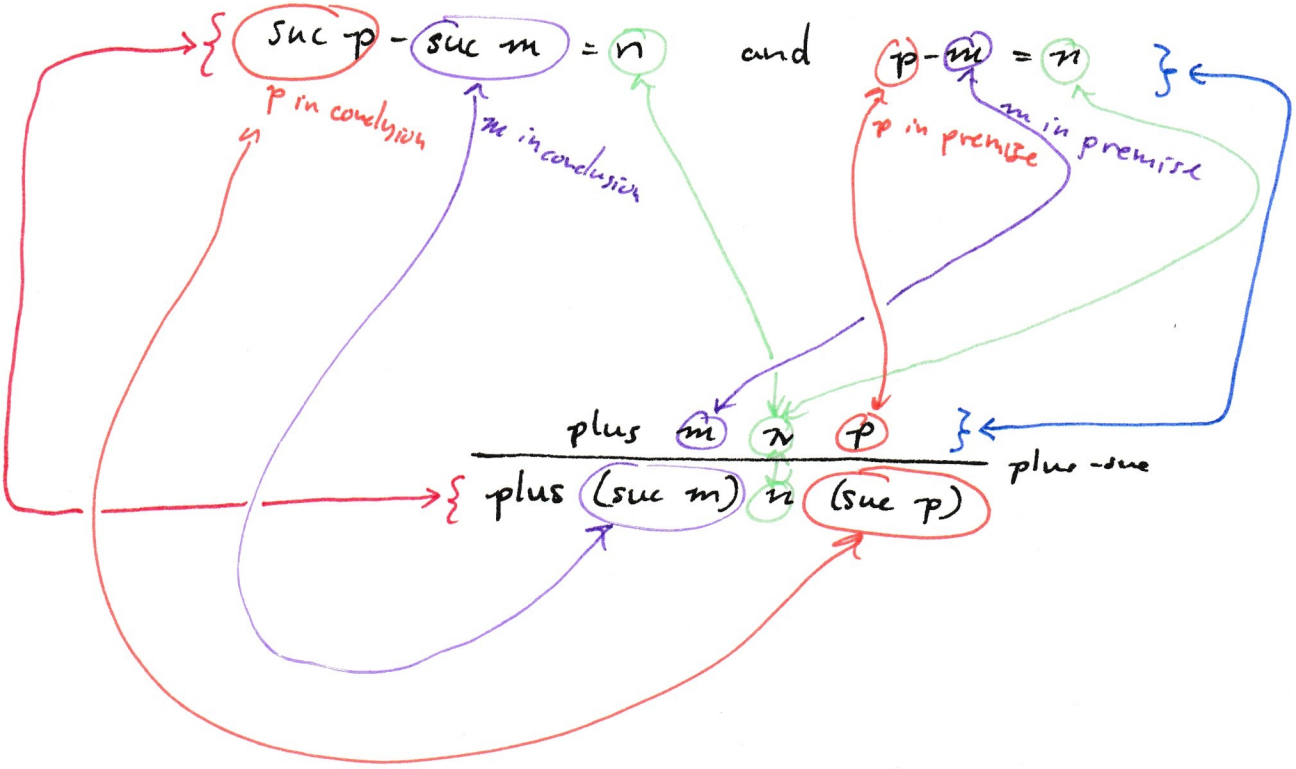
Looking back at our inference rules, we can see that the defined cases of subtraction correspond partially to each rule:





$$suc\ p - suc\ m = p - m$$

$$\Downarrow \text{equivalently}$$



Given that we have this clean correspondence between plus m n p's provability and the defined of $p - m = n$, we can confidently say that the implementation of plus should have <u>some</u> valid ordering of equations that correctly corresponds to provable judgments. Now we just need to determine what that actually is.

The route we'll take is to make our equations more like the defined cases of subtraction / provable cases of plus m n p. First, we'll split the equation for plus-0 into two equivalent equations:

```
plus n p = if n == p then Just 0 else Nothing
plus n (suc p) = case plus n p of
                   Nothing → Nothing
                   Just m → Just (suc m)
```


```
plus n 0 = if n == 0 then Just 0 else Nothing
plus n (suc p) = if n = suc p then Just 0 else Nothing
plus n (suc p) = case plus n p of
                   Nothing → Nothing
                   Just m → Just (suc m)
```

Next, we fuse the two equations for plus with the same RHS:

```
plus n 0 = if n == 0 then Just 0 else Nothing
plus n (suc p) = if n == suc p then Just 0
                 else case plus n p of
                        Nothing → Nothing
                        Just m → Just (suc m)
```

In the case of plus m n p, the return type is a monadic type, so we can dramatically improve readability by shifting to do-notation and other monadic operations:

```
plus n Ø = do guard (n == Ø)
              return Ø
plus n (Suc p) = if n == Suc p
                 then return Ø
                 else do m ← plus n p
                         return (Suc m)
```

Even the original plus implementation, which computed the sum rather than the difference, can be translated to do-notation with the Id monad:

```
plus :: Nat → Nat → Id Nat
plus Ø n = return n
plus (Suc m) n = do p ← plus m n
                    return (Suc p)
```

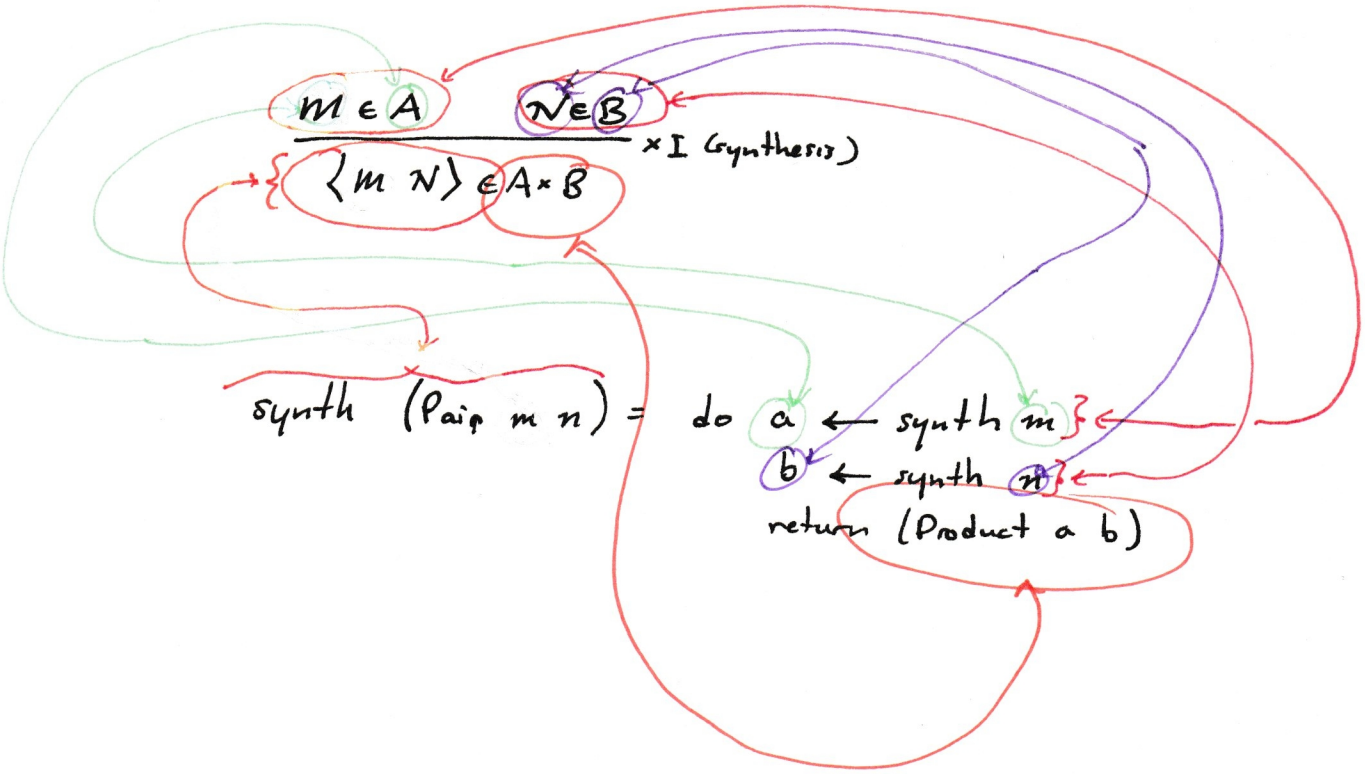
In this case, monadic bind acts almost identical to the original let expression:

```
plus (Suc m) n = let p = plus m n
                 in Suc p
```

In monadic notation, the mapping from proof rules to implementations is even easier:

1. Conclusions become LHSes just like before,
2. Premises become different bind statements in a do block, ordered so that the outputs of one judgment can flow into the inputs of other judgments.

Here's what that looks like for the pair synthesis rule ×I:



EXERCISE

Translate the rest of the bidirectional rules for pairs.