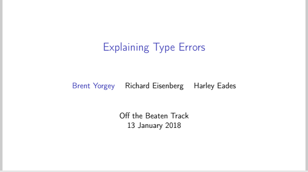
Explaining Type Errors

Brent Yorgey Richard Eisenberg Harley Eades

Off the Beaten Track
13 January 2018

Explaining Type Errors


Explaining Type Errors

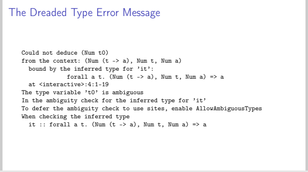
Brent Yorgey Richard Eisenberg Harley Eades

Off the Beaten Track
13 January 2018

Every beginning programmer using a statically typed language is all too familiar with. . .

The Dreaded Type Error Message

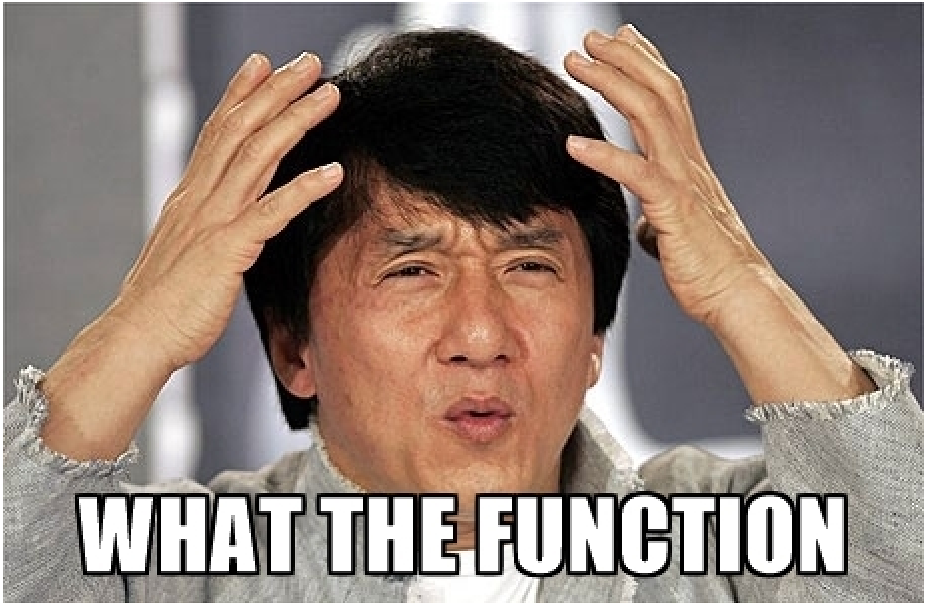
```
Could not deduce (Num t0)
from the context: (Num (t -> a), Num t, Num a)
  bound by the inferred type for ’it’:
              forall a t. (Num (t -> a), Num t, Num a) => a
  at <interactive>:4:1-19
The type variable ’t0’ is ambiguous
In the ambiguity check for the inferred type for ’it’
To defer the ambiguity check to use sites, enable AllowAmbiguousTypes
When checking the inferred type
  it :: forall a t. (Num (t -> a), Num t, Num a) => a
```



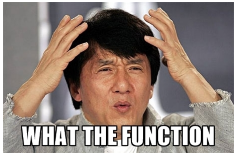
The Dreaded Type Error Message

```
Could not deduce (Num t0)
from the context: (Num (t -> a), Num t, Num a)
  bound by the inferred type for ’it’:
             forall a t. (Num (t -> a), Num t, Num a) => a
  at <interactive>:4:1-19
The type variable ’t0’ is ambiguous
In the ambiguity check for the inferred type for ’it’
To defer the ambiguity check to use sites, enable AllowAmbiguousTypes
When checking the inferred type
  it :: forall a t. (Num (t -> a), Num t, Num a) => a
```

the Dreaded Type Error Message!


WHAT THE FUNCTION

Explaining Type Errors



What can we do to make this better?

Theses

- "Improving" error messages doesn't fundamentally help.
- Interactive **error explanations** instead of static **error messages**.
- **Error explanation** = **constructive evidence** for an error.

Theses

- "Improving" error messages doesn't fundamentally help.
- Interactive **error explanations** instead of static **error messages**.
- **Error explanation** = **constructive evidence** for an error.

I'm going to propose three interrelated theses: first, although improving error messages is certainly worthwhile, it doesn't really fix the fundamental problem. Second, we should think about moving towards interactive error explanations rather than static error messages; finally, I will propose a framework for thinking about how to construct such explanations, in terms of constructive evidence for errors.

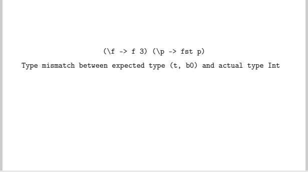
First, let's understand what the fundamental problem is, which is something I call "the curse of information".

The Curse of Information

```
(\f -> f 3) (\p -> fst p)
```

```
(\f -> f 3) (\p -> fst p)
```

```
Type mismatch between expected type (t, b0) and actual type Int
```



```
(\f -> f 3) (\p -> fst p)
Type mismatch between expected type (t, b0) and actual type Int
```

Suppose a hypothetical beginning programmer has written this expression. (This looks Haskell-ish but what I'm going to say isn't specific to any particular programming language.)
As it turns out, this is not type correct, so they might get an error message like this: apparently the type checker was expecting some kind of pair type but got an Int.
Now, for an experienced programmer, this might be enough to find and fix the error. But it's certainly not enough for our beginning programmer; the error message doesn't even say **where** the problem is.

```
(\f -> f 3) (\p -> fst p)
```

```
Type mismatch between expected type (t, b0) and actual type Int
  In the first argument of fst, namely p
  In the expression: fst p
  In the first argument of \ f -> f 3, namely
    (\ p -> fst p)
```

```
(\f -> f 3) (\p -> fst p)
```

```
Type mismatch between expected type (t, b0) and actual type Int
  In the first argument of fst, namely p
  In the expression: fst p
  In the first argument of \ f -> f 3, namely
    (\ p -> fst p)
  Inferred types for subterms:
    3              :: Int
    (\f -> f 3)    :: forall a. (Int -> a) -> a
    (\p -> fst p) :: (Int -> a0)
    (\f -> f 3) (\p -> fst p) :: a0
```

```
(\f -> f 3) (\p -> fst p)
```

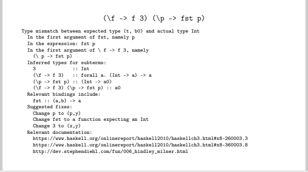
```
Type mismatch between expected type (t, b0) and actual type Int
  In the first argument of fst, namely p
  In the expression: fst p
  In the first argument of \ f -> f 3, namely
    (\ p -> fst p)
  Inferred types for subterms:
    3              :: Int
    (\f -> f 3)    :: forall a. (Int -> a) -> a
    (\p -> fst p) :: (Int -> a0)
    (\f -> f 3) (\p -> fst p) :: a0
  Relevant bindings include:
    fst :: (a,b) -> a
```

(\f -> f 3) (\p -> fst p)

```
Type mismatch between expected type (t, b0) and actual type Int
  In the first argument of fst, namely p
  In the expression: fst p
  In the first argument of \ f -> f 3, namely
    (\ p -> fst p)
  Inferred types for subterms:
    3             :: Int
    (\f -> f 3)   :: forall a. (Int -> a) -> a
    (\p -> fst p) :: (Int -> a0)
    (\f -> f 3) (\p -> fst p) :: a0
  Relevant bindings include:
    fst :: (a,b) -> a
  Suggested fixes:
    Change p to (p,y)
    Change fst to a function expecting an Int
    Change 3 to (x,y)
```

`(\f -> f 3) (\p -> fst p)`

```
Type mismatch between expected type (t, b0) and actual type Int
  In the first argument of fst, namely p
  In the expression: fst p
  In the first argument of \ f -> f 3, namely
    (\ p -> fst p)
  Inferred types for subterms:
    3             :: Int
    (\f -> f 3)   :: forall a. (Int -> a) -> a
    (\p -> fst p) :: (Int -> a0)
    (\f -> f 3) (\p -> fst p) :: a0
  Relevant bindings include:
    fst :: (a,b) -> a
  Suggested fixes:
    Change p to (p,y)
    Change fst to a function expecting an Int
    Change 3 to (x,y)
  Relevant documentation:
    https://www.haskell.org/onlinereport/haskell2010/haskellch3.html#x8-260003.3
    https://www.haskell.org/onlinereport/haskell2010/haskellch3.html#x8-360003.8
    http://dev.stephendiehl.com/fun/006_hindley_milner.html
```



```
(\f -> f 3) (\p -> fst p)
```

```
Type mismatch between expected type (t, b0) and actual type Int
  In the first argument of fst, namely p
  In the expression: fst p
  In the first argument of \ f -> f 3, namely
    (\ p -> fst p)
  Inferred types for subterms:
    3             :: Int
    (\f -> f 3)   :: forall a. (Int -> a) -> a
    (\p -> fst p) :: (Int -> a0)
    (\f -> f 3) (\p -> fst p) :: a0
  Relevant bindings include:
    fst :: (a,b) -> a
  Suggested fixes:
    Change p to (p,y)
    Change fst to a function expecting an Int
    Change 3 to (x,y)
  Relevant documentation:
    https://www.haskell.org/onlinereport/haskell2010/haskellch3.html#x8-260003.3
    https://www.haskell.org/onlinereport/haskell2010/haskellch3.html#x8-360003.8
    http://dev.stephendiehl.com/fun/006_hindley_milner.html
```

OK, so let's add more information! Now the error message says where the problem is.

But the beginning programmer still might not understand **why** there is an error. So let's add information about types of inferred subterms, so they can see where different types are coming from. But they might forget what `fst` is, so we could add information about it. Maybe they still have no idea what to do so we could add some suggested fixes. . . and links to relevant documentation. . .

YOU'RE NOT HELPING
COMEDY CENTRAL

This actually doesn't help! Why not?

The Curse of Information

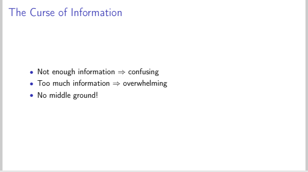
- Not enough information $\Rightarrow$ confusing

The Curse of Information

- Not enough information $\Rightarrow$ confusing
- Too much information $\Rightarrow$ overwhelming

The Curse of Information

- Not enough information $\Rightarrow$ confusing
- Too much information $\Rightarrow$ overwhelming
- No middle ground!



The Curse of Information

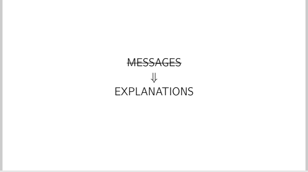
- Not enough information ⇒ confusing
- Too much information ⇒ overwhelming
- No middle ground!

This is what I am calling the Curse of Information. If there's not enough information, the programmer will obviously be confused and have no idea what is going on. On the other hand, if there is too **much** information, it will be overwhelming: both because much of the information may turn out to be irrelevant, so it's hard to pick out the information that is really needed; and simply because psychologically it is overwhelming to see a giant wall of text.
To make things worse, though, there **is no** middle ground! The problem is that the right amount of information, and which information is relevant, will vary from programmer to programmer and even from error to error with the same programmer.

~~MESSAGES~~

$\Downarrow$

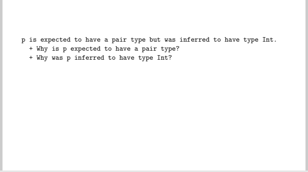
EXPLANATIONS



~~MESSAGES~~
⇓
EXPLANATIONS

The real problem is that we are fixated on static **error messages**. We ought to instead think about dynamic **error explanations** where the programmer gets to interactively pick exactly the information that is relevant to them.

```
p is expected to have a pair type but was inferred to have type Int.
  + Why is p expected to have a pair type?
  + Why was p inferred to have type Int?
```



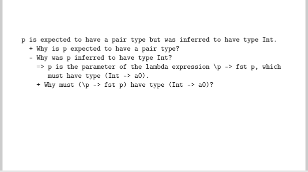
```
p is expected to have a pair type but was inferred to have type Int.
  + Why is p expected to have a pair type?
  + Why was p inferred to have type Int?
```

Let's look at a simple, completely made-up example of what this might look like for our running example. The programmer would initially be presented with a basic type mismatch message, together with several **questions** they can expand if they wish to see the answer.

In this case, perhaps the programmer thinks, "I definitely know why p is expected to have a pair type, because it is an argument to `fst`; what I don't understand is why it was inferred to have type `Int`."

So they expand that question.

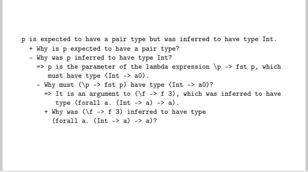
```
p is expected to have a pair type but was inferred to have type Int.
  + Why is p expected to have a pair type?
  - Why was p inferred to have type Int?
    => p is the parameter of the lambda expression \p -> fst p, which
       must have type (Int -> a0).
    + Why must (\p -> fst p) have type (Int -> a0)?
```



```
p is expected to have a pair type but was inferred to have type Int.
  + Why is p expected to have a pair type?
  - Why was p inferred to have type Int?
    => p is the parameter of the lambda expression \p -> fst p, which
       must have type (Int -> a0).
    + Why must (\p -> fst p) have type (Int -> a0)?
```

It might then explain to them that this is because p is the parameter of a lambda expression which must have a type whose domain is Int. Perhaps they don’t understand that either, so they can expand another question.

```
p is expected to have a pair type but was inferred to have type Int.
  + Why is p expected to have a pair type?
  - Why was p inferred to have type Int?
    => p is the parameter of the lambda expression \p -> fst p, which
       must have type (Int -> a0).
    - Why must (\p -> fst p) have type (Int -> a0)?
      => It is an argument to (\f -> f 3), which was inferred to have
         type (forall a. (Int -> a) -> a).
      + Why was (\f -> f 3) inferred to have type
        (forall a. (Int -> a) -> a)?
```



```
p is expected to have a pair type but was inferred to have type Int.
  + Why is p expected to have a pair type?
  - Why was p inferred to have type Int?
    => p is the parameter of the lambda expression \p -> fst p, which
       must have type (Int -> a0).
    - Why must (\p -> fst p) have type (Int -> a0)?
      => It is an argument to (\f -> f 3), which was inferred to have
         type (forall a. (Int -> a) -> a).
      + Why was (\f -> f 3) inferred to have type
        (forall a. (Int -> a) -> a)?
```

This step is then explained in turn: because this lambda expression is an argument to `(\f -> f 3)`, which was inferred to have a certain type. Perhaps, hypothetically, at this point the light bulb turns on and they don't need to expand any further.
[Something to point out, which I didn't say in the talk but fielded a question about later: I am not advocating for a textual question-answer format like this in particular. This is just one particular example of a possible manifestation of interactive error explanations. One could also imagine things involving graph visualizations, tooltips, or some mixture of all these things.]

Related work. . .

- Plociniczak & Odersky: Scalad (2012)
- Stuckey, Sulzmann & Wazny: Chameleon (2003)
- Simon, Chitil, & Huch (2000)
- Beaven & Stansifer (1993)

Not new! But not enough attention. . .

Related work. . .

- Plociniczak & Odersky: Scalad (2012)
- Stuckey, Sulzmann & Wazny: Chameleon (2003)
- Simon, Chitil, & Huch (2000)
- Beaven & Stansifer (1993)

Not new! But not enough attention. . .

This idea is not new. There has been work on related things over many years. Most recently, an interactive type debugger for Scala; in early 2000s there was a similar system for Haskell; in the 1990's there was some more foundational work. But in my opinion this area is not receiving enough attention.

[I did not mention this in my talk for time reasons, but some reviewers mentioned Seidel, Jhala & Weimer on generating dynamic witnesses for type errors (ICFP 2016). This is a really cool idea, but orthogonal to my proposal; we should do both.]

As far as I understand, all of these work by allowing the user to interactively explore typing derivations; if there is anything novel in my talk, it is my proposal of an alternative framework for thinking about how to construct error explanations.

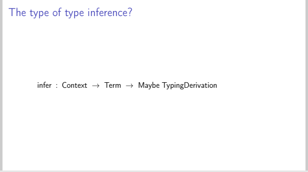
Explaining errors

The type of type inference?

$$\text{infer} : \text{Context} \rightarrow \text{Term} \rightarrow \text{Maybe Type}$$

The type of type inference?

$$\mathsf{infer} : \mathsf{Context} \rightarrow \mathsf{Term} \rightarrow \mathsf{Maybe\ TypingDerivation}$$



The type of type inference?

```
infer : Context → Term → Maybe TypingDerivation
```

Let's start by thinking about the type of a type inference algorithm. (One could tell a similar story for type checking but inference will be simpler for my purpose.) We could start with a simplistic version that takes as input a context and a term, and either outputs a type for the term or fails.

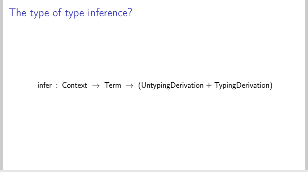
Of course, this is unsatisfactory: how do we know that the output type has anything to do with the input term? And we'd like to know **why** the given term has this type. The solution to this is well-known: instead of outputting just a type, we output a **typing derivation** which is a (constructive) proof that the given term has some type in the given context.

The type of type inference?

$$\mathsf{infer} \ : \ \mathsf{Context} \ \rightarrow \ \mathsf{Term} \ \rightarrow \ (\mathsf{Error} + \mathsf{TypingDerivation})$$

The type of type inference?

$$\mathsf{infer} \;:\; \mathsf{Context} \;\rightarrow\; \mathsf{Term} \;\rightarrow\; (\mathsf{UntypingDerivation} + \mathsf{TypingDerivation})$$


The type of type inference?

infer : Context → Term → (UntypingDerivation + TypingDerivation)

Of course, we don't just want to fail—we should return some kind of error if the term does not have a type. This is how lots of existing typecheckers actually look.

But simply generating an error is unsatisfactory for similar reasons that simply generating a type was unsatisfactory—how do we know the error has anything to do with the term? **Why** was a particular error generated?

The solution is also parallel: instead of an error we should return **constructive evidence** that the term does **not** have a type, which I call an **untyping derivation**.

The type of type inference?

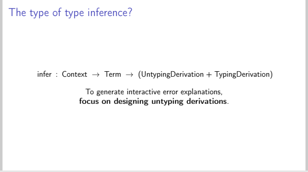
$$\text{infer} : \text{Context} \to \text{Term} \to (\text{UntypingDerivation} + \text{TypingDerivation})$$

See Ulf Norell keynote @ ICFP 2013:
`http://www.cse.chalmers.se/~ulfn/code/icfp2013/ICFP.html`

The type of type inference?

$$\text{infer} : \text{Context} \rightarrow \text{Term} \rightarrow (\text{UntypingDerivation} + \text{TypingDerivation})$$

To generate interactive error explanations,
focus on designing untyping derivations.



The type of type inference?

infer : Context → Term → (UntypingDerivation + TypingDerivation)

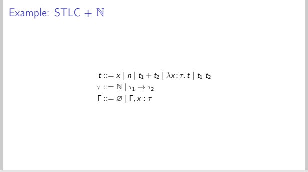
To generate interactive error explanations,
focus on designing untyping derivations.

This is not really new either: Ulf Norell actually gave a nice keynote at ICFP in Boston where he essentially livecoded a type inference algorithm very much like this for the STLC in Agda.

I propose that a principled way to think about generating error explanations is to focus on designing untyping derivations.

Example: STLC + $\mathbb{N}$

$$
\begin{aligned}
t &::= x \mid n \mid t_1 + t_2 \mid \lambda x\!:\!\tau.\, t \mid t_1\ t_2 \\
\tau &::= \mathbb{N} \mid \tau_1 \to \tau_2 \\
\Gamma &::= \varnothing \mid \Gamma, x : \tau
\end{aligned}
$$



Example: STLC + $\mathbb{N}$

$$\begin{aligned} t &::= x \mid n \mid t_1 + t_2 \mid \lambda x{:}\tau.\, t \mid t_1\ t_2 \\ \tau &::= \mathbb{N} \mid \tau_1 \to \tau_2 \\ \Gamma &::= \varnothing \mid \Gamma, x : \tau \end{aligned}$$

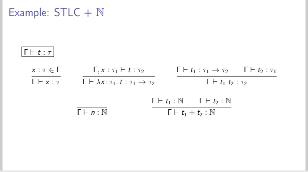
Let's look at a simple example. We'll consider the STLC with natural number literals and addition expressions. Notice that lambdas have type annotations which will make things a lot simpler. There is a primitive type of natural numbers and arrow types.

Example: STLC + $\mathbb{N}$

$$\boxed{\Gamma \vdash t : \tau}$$

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.\, t : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash t_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1\ t_2 : \tau_2}$$

$$\frac{}{\Gamma \vdash n : \mathbb{N}} \qquad \frac{\Gamma \vdash t_1 : \mathbb{N} \qquad \Gamma \vdash t_2 : \mathbb{N}}{\Gamma \vdash t_1 + t_2 : \mathbb{N}}$$



Example: STLC + $\mathbb{N}$

$$\boxed{\Gamma \vdash t : \tau}$$

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x{:}\tau_1 .\, t : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash t_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1\ t_2 : \tau_2}$$

$$\frac{}{\Gamma \vdash n : \mathbb{N}} \qquad \frac{\Gamma \vdash t_1 : \mathbb{N} \qquad \Gamma \vdash t_2 : \mathbb{N}}{\Gamma \vdash t_1 + t_2 : \mathbb{N}}$$

And here is the type system; this is entirely standard.

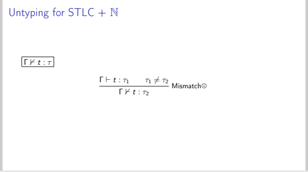
Untyping for STLC + $\mathbb{N}$

$$\boxed{\Gamma \nvdash t : \tau}$$

Untyping for STLC + $\mathbb{N}$

$$\boxed{\Gamma \nvdash t : \tau}$$

$$\frac{\Gamma \vdash t : \tau_1 \qquad \tau_1 \neq \tau_2}{\Gamma \nvdash t : \tau_2}\ \text{Mismatch☹}$$


Untyping for STLC + $\mathbb{N}$

$\boxed{\Gamma \nvdash t : \tau}$

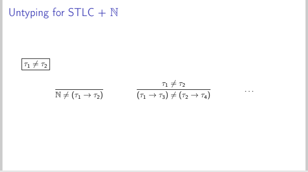
$$\frac{\Gamma \vdash t : \tau_1 \qquad \tau_1 \neq \tau_2}{\Gamma \nvdash t : \tau_2}\ \text{Mismatch☹}$$

So, let's think about how to design **untyping** derivations for this language. There are actually a lot of ways to do this, I'm just going to show one example.
Here's the first rule I will propose, which is fairly simple: if t has some type τ_1, then it does not have some different type τ_2. Of course, this only works because the STLC has unique types; if you had a system without unique types then you wouldn't have this rule.
There are a few things to point out. One is that of course this rule references the typing judgment, which is probably typical.
Another thing to point out is about the other premise, $\tau_1 \neq \tau_2$: this is another negative, but we don't just want it to be the negation of equality; we want positive evidence that τ_1 and τ_2 are different, which we can use to explain **why** they are different.

Untyping for STLC + $\mathbb{N}$

$$\boxed{\tau_1 \neq \tau_2}$$

$$\frac{}{\mathbb{N} \neq (\tau_1 \to \tau_2)} \qquad \frac{\tau_1 \neq \tau_2}{(\tau_1 \to \tau_3) \neq (\tau_2 \to \tau_4)} \qquad \cdots$$


Untyping for STLC + $\mathbb{N}$

$\boxed{\tau_1 \neq \tau_2}$

$$\frac{}{\mathbb{N} \neq (\tau_1 \to \tau_2)} \qquad \frac{\tau_1 \neq \tau_2}{(\tau_1 \to \tau_3) \neq (\tau_2 \to \tau_4)} \qquad \cdots$$

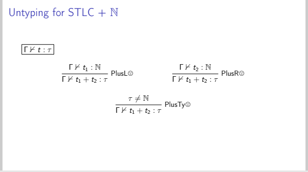
For example, we'd probably have some rules like this: Nat is not an arrow type; some congruence rules; and so on.

Untyping for STLC + $\mathbb{N}$

$\boxed{\Gamma \nvdash t : \tau}$

$$\frac{\Gamma \nvdash t_1 : \mathbb{N}}{\Gamma \nvdash t_1 + t_2 : \tau}\ \text{PlusL☹} \qquad \frac{\Gamma \nvdash t_2 : \mathbb{N}}{\Gamma \nvdash t_1 + t_2 : \tau}\ \text{PlusR☹}$$

$$\frac{\tau \neq \mathbb{N}}{\Gamma \nvdash t_1 + t_2 : \tau}\ \text{PlusTy☹}$$



Untyping for STLC + ℕ

$\boxed{\Gamma \nvdash t : \tau}$

$$\frac{\Gamma \nvdash t_1 : \mathbb{N}}{\Gamma \nvdash t_1 + t_2 : \tau}\ \text{PlusL☹} \qquad \frac{\Gamma \nvdash t_2 : \mathbb{N}}{\Gamma \nvdash t_1 + t_2 : \tau}\ \text{PlusR☹}$$

$$\frac{\tau \neq \mathbb{N}}{\Gamma \nvdash t_1 + t_2 : \tau}\ \text{PlusTy☹}$$

Now for some rules about addition. There are basically two ways an addition expression could fail to have a particular type. One is if the type is not ℕ. The other is if one of the two subterms does not have type ℕ.

Again, there are other ways we could encode this. Part of the point is that we have some freedom in choosing rules that will result in the sort of explanations we want.

Untyping for STLC + $\mathbb{N}$

$$\boxed{\Gamma \nvdash t : \tau}$$

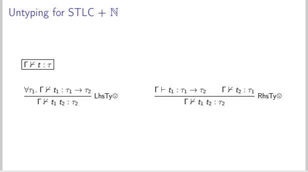
$$\frac{\forall \tau_2.\ \tau \neq (\tau_1 \to \tau_2)}{\Gamma \nvdash \lambda x{:}\tau_1.\ t : \tau}\ \text{AbsTy☹} \qquad \frac{\Gamma, x : \tau_1 \nvdash t : \tau_2}{\Gamma \nvdash \lambda x{:}\tau_1.\ t : \tau_1 \to \tau_2}\ \text{AbsBody☹}$$

Untyping for STLC + $\mathbb{N}$

$$\boxed{\Gamma \nvdash t : \tau}$$

$$\frac{\forall \tau_1.\ \Gamma \nvdash t_1 : \tau_1 \to \tau_2}{\Gamma \nvdash t_1\ t_2 : \tau_2}\ \text{LhsTy☹}$$

$$\frac{\Gamma \vdash t_1 : \tau_1 \to \tau_2 \qquad \Gamma \nvdash t_2 : \tau_1}{\Gamma \nvdash t_1\ t_2 : \tau_2}\ \text{RhsTy☹}$$



Untyping for STLC + ℕ

$\boxed{\Gamma \nvdash t : \tau}$

$$\frac{\forall \tau_1.\ \Gamma \nvdash t_1 : \tau_1 \to \tau_2}{\Gamma \nvdash t_1\ t_2 : \tau_2}\ \text{LhsTy☹} \qquad \frac{\Gamma \vdash t_1 : \tau_1 \to \tau_2 \qquad \Gamma \nvdash t_2 : \tau_1}{\Gamma \nvdash t_1\ t_2 : \tau_2}\ \text{RhsTy☹}$$

Then we have some rules about lambdas. There are two ways a lambda expression can fail to have type τ. The first is if τ is not an arrow type with the correct domain. Otherwise, if τ is an arrow type with a matching domain, the body could fail to have the type of the codomain.

I'll skip over the rules for function application since I won't use them in my examples.

Example

Does $\lambda f : \mathbb{N} \to \mathbb{N}.\, f + 2$ have type $(\mathbb{N} \to \mathbb{N}) \to \mathbb{N} \to \mathbb{N}$?

Example

Does $\lambda f : \mathbb{N} \to \mathbb{N}.\, f + 2$ have type $(\mathbb{N} \to \mathbb{N}) \to \mathbb{N} \to \mathbb{N}$?

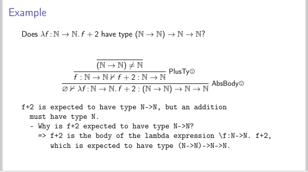
$$\dfrac{\dfrac{\dfrac{}{(\mathbb{N} \to \mathbb{N}) \neq \mathbb{N}}}{f : \mathbb{N} \to \mathbb{N} \nvdash f + 2 : \mathbb{N} \to \mathbb{N}}\ \text{PlusTy☹}}{\varnothing \nvdash \lambda f : \mathbb{N} \to \mathbb{N}.\, f + 2 : (\mathbb{N} \to \mathbb{N}) \to \mathbb{N} \to \mathbb{N}}\ \text{AbsBody☹}$$

Example

Does $\lambda f : \mathbb{N} \to \mathbb{N}.\, f + 2$ have type $(\mathbb{N} \to \mathbb{N}) \to \mathbb{N} \to \mathbb{N}$?

$$\dfrac{\dfrac{\dfrac{}{(\mathbb{N} \to \mathbb{N}) \neq \mathbb{N}}}{f : \mathbb{N} \to \mathbb{N} \nvdash f + 2 : \mathbb{N} \to \mathbb{N}}\ \text{PlusTy☹}}{\varnothing \nvdash \lambda f : \mathbb{N} \to \mathbb{N}.\, f + 2 : (\mathbb{N} \to \mathbb{N}) \to \mathbb{N} \to \mathbb{N}}\ \text{AbsBody☹}$$

```
f+2 is expected to have type N->N, but an addition
  must have type N.
  - Why is f+2 expected to have type N->N?
    => f+2 is the body of the lambda expression \f:N->N. f+2,
       which is expected to have type (N->N)->N->N.
```



Example

Does $\lambda f : \mathbb{N} \to \mathbb{N}.\, f + 2$ have type $(\mathbb{N} \to \mathbb{N}) \to \mathbb{N} \to \mathbb{N}$?

$$\dfrac{\dfrac{\overline{(\mathbb{N} \to \mathbb{N}) \neq \mathbb{N}}}{f : \mathbb{N} \to \mathbb{N} \nvdash f + 2 : \mathbb{N} \to \mathbb{N}}\ \text{PlusTy☹}}{\varnothing \nvdash \lambda f : \mathbb{N} \to \mathbb{N}.\, f + 2 : (\mathbb{N} \to \mathbb{N}) \to \mathbb{N} \to \mathbb{N}}\ \text{AbsBody☹}$$

```
f+2 is expected to have type N->N, but an addition
  must have type N.
  - Why is f+2 expected to have type N->N?
    => f+2 is the body of the lambda expression \f:N->N. f+2,
       which is expected to have type (N->N)->N->N.
```

Let's look at an example. Consider asking whether this lambda expression has this particular type. If you think about it for a bit you can see that it does not, but how do we formally show it?
Here's one untyping derivation we could give. The type **is** in fact an arrow type with the correct domain, so the final rule has to be AbsBody. At that point we note that an addition expression cannot have type $\mathbb{N} \to \mathbb{N}$ since it is not equal to $\mathbb{N}$.
And here is a possible explanation that could be generated from this derivation. Note how each statement corresponds to a rule in the derivation.

Example, take 2

Does $\lambda f : \mathbb{N} \to \mathbb{N}.\, f + 2$ have type $(\mathbb{N} \to \mathbb{N}) \to \mathbb{N} \to \mathbb{N}$?

Example, take 2

Does $\lambda f:\mathbb{N}\to\mathbb{N}.\,f+2$ have type $(\mathbb{N}\to\mathbb{N})\to\mathbb{N}\to\mathbb{N}$?

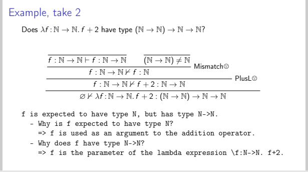
$$\dfrac{\dfrac{\dfrac{\dfrac{}{f:\mathbb{N}\to\mathbb{N}\vdash f:\mathbb{N}\to\mathbb{N}}\qquad\dfrac{}{(\mathbb{N}\to\mathbb{N})\neq\mathbb{N}}}{f:\mathbb{N}\to\mathbb{N}\nvdash f:\mathbb{N}}\,\text{Mismatch☹}}{f:\mathbb{N}\to\mathbb{N}\nvdash f+2:\mathbb{N}\to\mathbb{N}}\,\text{PlusL☹}}{\varnothing\nvdash \lambda f:\mathbb{N}\to\mathbb{N}.\,f+2:(\mathbb{N}\to\mathbb{N})\to\mathbb{N}\to\mathbb{N}}$$

Example, take 2

Does $\lambda f : \mathbb{N} \to \mathbb{N}.\, f + 2$ have type $(\mathbb{N} \to \mathbb{N}) \to \mathbb{N} \to \mathbb{N}$?

$$
\dfrac{
  \dfrac{
    \dfrac{
      \dfrac{}{f : \mathbb{N} \to \mathbb{N} \vdash f : \mathbb{N} \to \mathbb{N}}
      \qquad
      \dfrac{}{(\mathbb{N} \to \mathbb{N}) \neq \mathbb{N}}
    }{f : \mathbb{N} \to \mathbb{N} \nvdash f : \mathbb{N}}\ \text{Mismatch☹}
  }{f : \mathbb{N} \to \mathbb{N} \nvdash f + 2 : \mathbb{N} \to \mathbb{N}}\ \text{PlusL☹}
}{\varnothing \nvdash \lambda f : \mathbb{N} \to \mathbb{N}.\, f + 2 : (\mathbb{N} \to \mathbb{N}) \to \mathbb{N} \to \mathbb{N}}
$$

```
f is expected to have type N, but has type N->N.
  - Why is f expected to have type N?
    => f is used as an argument to the addition operator.
  - Why does f have type N->N?
    => f is the parameter of the lambda expression \f:N->N. f+2.
```



Example, take 2

Does $\lambda f:\mathbb{N}\to\mathbb{N}.\, f+2$ have type $(\mathbb{N}\to\mathbb{N})\to\mathbb{N}\to\mathbb{N}$?

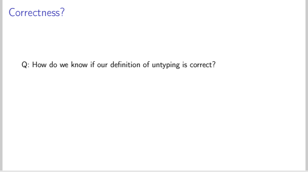
$$\dfrac{\dfrac{\dfrac{\overline{f:\mathbb{N}\to\mathbb{N}\vdash f:\mathbb{N}\to\mathbb{N}} \qquad \overline{(\mathbb{N}\to\mathbb{N})\neq\mathbb{N}}}{f:\mathbb{N}\to\mathbb{N}\nvdash f:\mathbb{N}}\ \text{Mismatch☹}}{f:\mathbb{N}\to\mathbb{N}\nvdash f+2:\mathbb{N}\to\mathbb{N}}\ \text{PlusL☹}}{\varnothing\nvdash \lambda f:\mathbb{N}\to\mathbb{N}.\, f+2:(\mathbb{N}\to\mathbb{N})\to\mathbb{N}\to\mathbb{N}}$$

```
f is expected to have type N, but has type N->N.
  - Why is f expected to have type N?
    => f is used as an argument to the addition operator.
  - Why does f have type N->N?
    => f is the parameter of the lambda expression \f:N->N. f+2.
```

There is actually a different derivation that could be given. It starts in the same way as before, but there is another reason that $f + 2$ is not well-typed, namely, that f does not have type $\mathbb{N}$.
And here is the explanation that might correspond to this. Notice that when the user expands the question as to why f has type $\mathbb{N}\to\mathbb{N}$, we must jump down in the derivation to the rule that put f into the context. So explanations do not necessarily simply step through the tree to adjacent nodes. One could imagine storing in the context alongside f a pointer to the node in the derivation which put f into the context.

Correctness?

Q: How do we know if our definition of untyping is correct?


Correctness?

Q: How do we know if our definition of untyping is correct?

So I have just finished showing you a bunch of rules for untyping derivations for the STLC. How can we have any confidence that these rules are the right ones, or I haven't left out any cases?

Correctness

A: prove a metatheorem!

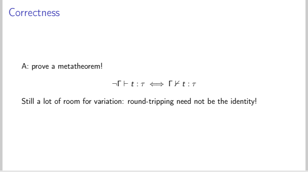
$$\neg \Gamma \vdash t : \tau \iff \Gamma \nvdash t : \tau$$

Correctness

A: prove a metatheorem!

$$\neg \Gamma \vdash t : \tau \iff \Gamma \nvdash t : \tau$$

Still a lot of room for variation: round-tripping need not be the identity!


Correctness

A: prove a metatheorem!

$$\neg \Gamma \vdash t : \tau \iff \Gamma \nvdash t : \tau$$

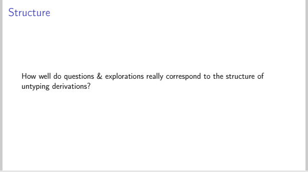
Still a lot of room for variation: round-tripping need not be the identity!

Well, untyping derivations are supposed to prove that a term does not have a certain type, so we should just prove a metatheorem showing that untyping is logically equivalent to the negation of typing.
I have in fact formalized the system I just showed you in Agda, and proved this metatheorem—in fact, through the process of proving it I fixed a few bugs in my rules!
Notice that this metatheorem does not constrain untyping to a single unique solution; in general there may be many different definitions of untyping which all satisfy this theorem. Intuitively, this is because round-tripping through this logical equivalence need not get us back to where we started.

Challenges

Structure

How well do questions & explorations really correspond to the structure of untyping derivations?


Structure

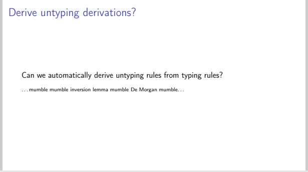
How well do questions & explorations really correspond to the structure of untyping derivations?

There are many remaining challenges; I have really only sketched an idea.
One challenge is simply to see how well this correspondence between untyping derivations and the kind of error explanations we want to generate scales up. I have only tried it for small toy languages so far.

Derive untyping derivations?

Can we automatically derive untyping rules from typing rules?

. . . mumble mumble inversion lemma mumble De Morgan mumble. . .


Derive untyping derivations?

Can we automatically derive untyping rules from typing rules?

. . . mumble mumble inversion lemma mumble De Morgan mumble. . .

Another challenge: can we automatically derive an untyping judgment from a typing judgment? Of course we do want the freedom to design untyping judgments in order to get the error explanations we want, but especially for larger systems it would be tedious to have to design all the rules by hand.

I have some very vague ideas about how to do this but could really only mumble about it at this point.

Unification?

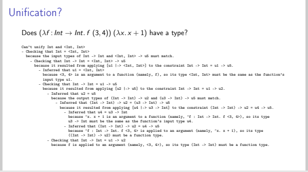
Unification?

Does $(\lambda f : \mathit{Int} \to \mathit{Int}.\, f\ (3, 4))\ (\lambda x.\, x + 1)$ have a type?

Unification?

Does $(\lambda f : Int \to Int.\, f\ (3, 4))\ (\lambda x.\, x + 1)$ have a type?

```
Can’t unify Int and <Int, Int>
- Checking that Int = <Int, Int>
  because the input types of Int -> Int and <Int, Int> -> u5 must match.
    - Checking that Int -> Int = <Int, Int> -> u5
      because it resulted from applying [u1 |-> <Int, Int>] to the constraint Int -> Int = u1 -> u5.
        - Inferred that u1 = <Int, Int>
          because <3, 4> is an argument to a function (namely, f), so its type <Int, Int> must be the same as the function’s
          input type u1.
        - Checking that Int -> Int = u1 -> u5
          because it resulted from applying [u2 |-> u5] to the constraint Int -> Int = u1 -> u2.
            - Inferred that u2 = u5
              because the output types of (Int -> Int) -> u2 and (u3 -> Int) -> u5 must match.
                - Inferred that (Int -> Int) -> u2 = (u3 -> Int) -> u5
                  because it resulted from applying [u4 |-> u3 -> Int] to the constraint (Int -> Int) -> u2 = u4 -> u5.
                    - Inferred that u4 = u3 -> Int
                      because ^x. x + 1 is an argument to a function (namely, ^f : Int -> Int. f <3, 4>), so its type
                      u3 -> Int must be the same as the function’s input type u4.
                    - Inferred that (Int -> Int) -> u2 = u4 -> u5
                      because ^f : Int -> Int. f <3, 4> is applied to an argument (namely, ^x. x + 1), so its type
                      ((Int -> Int) -> u2) must be a function type.
            - Checking that Int -> Int = u1 -> u2
              because f is applied to an argument (namely, <3, 4>), so its type (Int -> Int) must be a function type.
```



Unification?

Does $(\lambda f : \mathit{Int} \to \mathit{Int}.\, f\ (3, 4))\ (\lambda x.\, x + 1)$ have a type?

```
Can't unify Int and <Int, Int>
- Checking that Int = <Int, Int>
  because the input types of Int -> Int and <Int, Int> -> u5 must match.
    - Checking that Int -> Int = <Int, Int> -> u5
      because it resulted from applying [u1 |-> <Int, Int>] to the constraint Int -> Int = u1 -> u5.
        - Inferred that u1 = <Int, Int>
          because <3, 4> is an argument to a function (namely, f), so its type <Int, Int> must be the same as the function's
          input type u1.
        - Checking that Int -> Int = u1 -> u5
          because it resulted from applying [u2 |-> u5] to the constraint Int -> Int = u1 -> u2.
            - Inferred that u2 = u5
              because the output types of (Int -> Int) -> u2 and (u3 -> Int) -> u5 must match.
                - Inferred that (Int -> Int) -> u2 = (u3 -> Int) -> u5
                  because it resulted from applying [u4 |-> u3 -> Int] to the constraint (Int -> Int) -> u2 = u4 -> u5.
                    - Inferred that u4 = u3 -> Int
                      because ^x. x + 1 is an argument to a function (namely, ^f : Int -> Int. f <3, 4>), so its type
                      u3 -> Int must be the same as the function's input type u4.
                    - Inferred that (Int -> Int) -> u2 = u4 -> u5
                      because ^f : Int -> Int. f <3, 4> is applied to an argument (namely, ^x. x + 1), so its type
                      ((Int -> Int) -> u2) must be a function type.
            - Checking that Int -> Int = u1 -> u2
              because f is applied to an argument (namely, <3, 4>), so its type (Int -> Int) must be a function type.
```

Finally, what about systems where typing involves unification? I tried implementing full type reconstruction for a version of the STLC with no type annotations on lambdas via unification, which kept track of what it was doing so it could generate explanations when something went wrong.

Consider this simple example. It is not well-typed, and we would like an explanation that says something about *f* expecting an Int but being given a tuple. Unfortunately, this is the explanation that was generated! As you can see, most of it has to do with unification variables and substitutions and so on, and is very difficult to follow unless you already know how unification works.

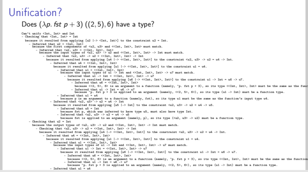
Unification?

Does $(\lambda p.\, \mathit{fst}\ p + 3)\ ((2,5),6)$ have a type?

Unification?

Does $(\lambda p.\, \mathit{fst}\ p + 3)\ ((2,5),6)$ have a type?

```
Can't unify <Int, Int> and Int
- Checking that <Int, Int> = Int
  because it resulted from applying [u2 |-> <Int, Int>] to the constraint u2 = Int.
    - Inferred that u2 = <Int, Int>
      because the first components of <u2, u3> and <<Int, Int>, Int> must match.
        - Inferred that <u2, u3> = <<Int, Int>, Int>
          because the input types of <u2, u3> -> u2 and <<Int, Int>, Int> -> Int must match.
            - Inferred that <u2, u3> -> u2 = <<Int, Int>, Int> -> Int
              because it resulted from applying [u4 |-> <<Int, Int>, Int>] to the constraint <u2, u3> -> u2 = u4 -> Int.
                - Inferred that u4 = <<Int, Int>, Int>
                  because it resulted from applying [u1 |-> <<Int, Int>, Int>] to the constraint u1 = u4.
                    - Inferred that u1 = <<Int, Int>, Int>
                      because the input types of u1 -> Int and <<Int, Int>, Int> -> u7 must match.
                        - Inferred that u1 -> Int = <<Int, Int>, Int> -> u7
                          because it resulted from applying [u6 |-> <<Int, Int>, Int>] to the constraint u1 -> Int = u6 -> u7.
                            - Inferred that u6 = <<Int, Int>, Int>
                              because <<2, 5>, 6> is an argument to a function (namely, ^p. fst p + 3), so its type <<Int, Int>, Int> must be the same as the fun
                            - Inferred that u1 -> Int = u6 -> u7
                              because ^p. fst p + 3 is applied to an argument (namely, <<2, 5>, 6>), so its type (u1 -> Int) must be a function type.
                    - Inferred that u1 = u4
                      because p is an argument to a function (namely, fst), so its type u1 must be the same as the function's input type u4.
                - Inferred that <u2, u3> -> u2 = u4 -> Int
                  because it resulted from applying [u5 |-> Int] to the constraint <u2, u3> -> u2 = u4 -> u5.
                    - Inferred that u5 = Int
                      because fst p, which was inferred to have type u5, must also have type Int.
                    - Inferred that <u2, u3> -> u2 = u4 -> u5
                      because fst is applied to an argument (namely, p), so its type (<u2, u3> -> u2) must be a function type.
    - Checking that u2 = Int
      because the output types of <u2, u3> -> u2 and <<Int, Int>, Int> -> Int must match.
        - Checking that <u2, u3> -> u2 = <<Int, Int>, Int> -> Int
          because it resulted from applying [u4 |-> <<Int, Int>, Int>] to the constraint <u2, u3> -> u2 = u4 -> Int.
            - Inferred that u4 = <<Int, Int>, Int>
              because it resulted from applying [u1 |-> <<Int, Int>, Int>] to the constraint u1 = u4.
                - Inferred that u1 = <<Int, Int>, Int>
                  because the input types of u1 -> Int and <<Int, Int>, Int> -> u7 must match.
                    - Inferred that u1 -> Int = <<Int, Int>, Int> -> u7
                      because it resulted from applying [u6 |-> <<Int, Int>, Int>] to the constraint u1 -> Int = u6 -> u7.
                        - Inferred that u6 = <<Int, Int>, Int>
                          because <<2, 5>, 6> is an argument to a function (namely, ^p. fst p + 3), so its type <<Int, Int>, Int> must be the same as the functio
                        - Inferred that u1 -> Int = u6 -> u7
                          because ^p. fst p + 3 is applied to an argument (namely, <<2, 5>, 6>), so its type (u1 -> Int) must be a function type.
                - Inferred that u1 = u4
```



Unification?

Does $(\lambda p.\ \mathit{fst}\ p + 3)\ ((2,5),6)$ have a type?

```
Can't unify <Int, Int> and Int
- Checking that <Int, Int> = Int
  because it resulted from applying [u2 |-> <Int, Int>] to the constraint u2 = Int.
  - Inferred that u2 = <Int, Int>
    because the first components of <u2, u3> and <<Int, Int>, Int> must match.
    - Inferred that <u2, u3> = <<Int, Int>, Int>
      because the input types of <u2, u3> -> u2 and <<Int, Int>, Int> -> Int must match.
      - Inferred that <u2, u3> -> u2 = <<Int, Int>, Int> -> Int
        because it resulted from applying [u4 |-> <<Int, Int>, Int>] to the constraint <u2, u3> -> u2 = u4 -> Int.
        - Inferred that u4 = <<Int, Int>, Int>
          because it resulted from applying [u1 |-> <<Int, Int>, Int>] to the constraint u1 = u4.
          - Inferred that u1 = <<Int, Int>, Int>
            because the input types of u1 -> Int and <<Int, Int>, Int> -> u7 must match.
            - Inferred that u1 -> Int = <<Int, Int>, Int> -> u7
              because it resulted from applying [u6 |-> <<Int, Int>, Int>] to the constraint u1 -> Int = u6 -> u7.
              - Inferred that u6 = <<Int, Int>, Int>
                because <<2, 5>, 6> is an argument to a function (namely, ~p. fst p + 3), so its type <<Int, Int>, Int> must be the same as the function's input type u6.
              - Inferred that u1 -> Int = u6 -> u7
                because ~p. fst p + 3 is applied to an argument (namely, <<2, 5>, 6>), so its type (u1 -> Int) must be a function type.
          - Inferred that u1 = u4
            because p is an argument to a function (namely, fst), so its type u1 must be the same as the function's input type u4.
        - Inferred that <u2, u3> -> u2 = u4 -> Int
          because it resulted from applying [u5 |-> Int] to the constraint <u2, u3> -> u2 = u4 -> u5.
          - Inferred that u5 = Int
            because fst p, which was inferred to have type u5, must also have type Int.
          - Inferred that <u2, u3> -> u2 = u4 -> u5
            because fst is applied to an argument (namely, p), so its type (<u2, u3> -> u2) must be a function type.
  - Checking that u2 = Int
    because the output types of <u2, u3> -> u2 and <<Int, Int>, Int> -> Int must match.
    - Checking that <u2, u3> -> u2 = <<Int, Int>, Int> -> Int
      because it resulted from applying [u4 |-> <<Int, Int>, Int>] to the constraint <u2, u3> -> u2 = u4 -> Int.
      - Inferred that u4 = <<Int, Int>, Int>
        because it resulted from applying [u1 |-> <<Int, Int>, Int>] to the constraint u1 = u4.
        - Inferred that u1 = <<Int, Int>, Int>
          because the input types of u1 -> Int and <<Int, Int>, Int> -> u7 must match.
          - Inferred that u1 -> Int = <<Int, Int>, Int> -> u7
            because it resulted from applying [u6 |-> <<Int, Int>, Int>] to the constraint u1 -> Int = u6 -> u7.
            - Inferred that u6 = <<Int, Int>, Int>
              because <<2, 5>, 6> is an argument to a function (namely, ~p. fst p + 3), so its type <<Int, Int>, Int> must be the same as the function's input type u6.
            - Inferred that u1 -> Int = u6 -> u7
              because ~p. fst p + 3 is applied to an argument (namely, <<2, 5>, 6>), so its type (u1 -> Int) must be a function type.
        - Inferred that u1 = u4
```

In fact, it gets much worse: this term is not much more complicated, and the error explanation does not even fit on the slide. Again, most of it has to do with unification variables and such.

Unification?

How to explain unification failures to the user?

Unification?

How to explain unification failures to the user?

- Implementation matters!

Unification?

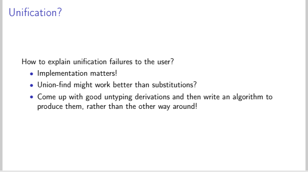
How to explain unification failures to the user?

- Implementation matters!
- Union-find might work better than substitutions?

Unification?

How to explain unification failures to the user?

- Implementation matters!
- Union-find might work better than substitutions?
- Come up with good untyping derivations and then write an algorithm to produce them, rather than the other way around!



Unification?

How to explain unification failures to the user?

- Implementation matters!
- Union-find might work better than substitutions?
- Come up with good untyping derivations and then write an algorithm to produce them, rather than the other way around!

I think there are a few lessons I learned from this. One is that the implementation matters! We are used to thinking of the implementation as being unimportant, except perhaps for efficiency. But actually it can make a big difference in terms of what sorts of explanations are easy to generate.
The fastest implementations of unification actually use a union-find structure rather than composing lots of substitutions; in this case, I have a vague intuition that a union-find structure might actually make things easier to explain.
More importantly, I think I did things sort of backwards: what I should have done was first design an untyping judgment that corresponds to the sort of explanations I would like to see, and then figure out how to produce them, rather than the other way around.

Questions/comments/ideas/discussion?