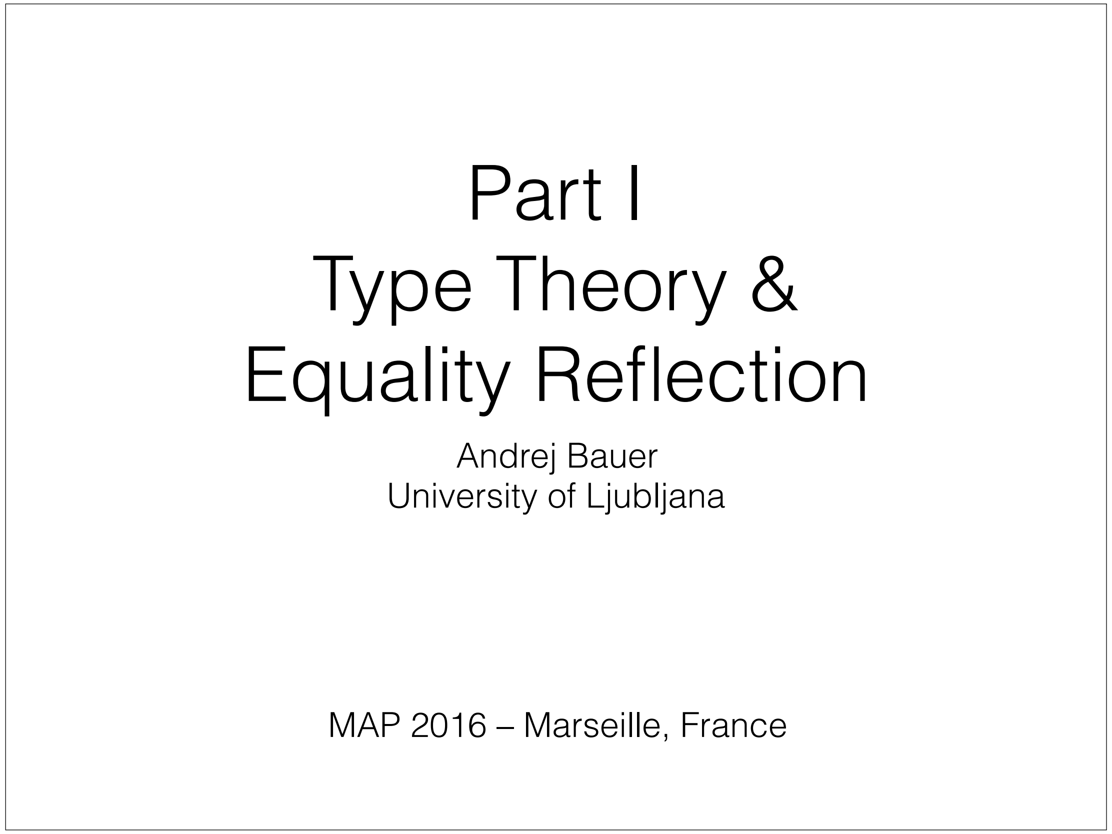


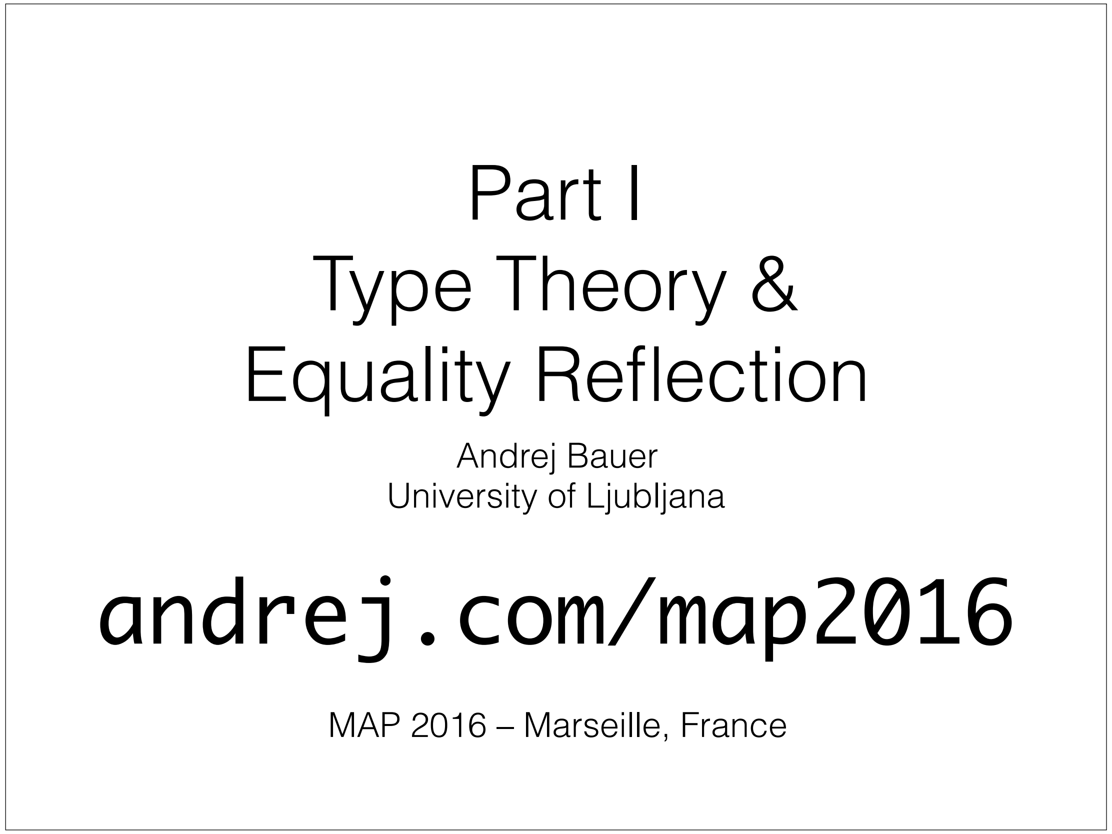
# Part I
# Type Theory & Equality Reflection

Andrej Bauer
University of Ljubljana

MAP 2016 – Marseille, France

Thank you for the invitation.
The title of this meeting is “Effective Analysis: Foundations, Implementations, Certification”. I will give three lectures, which fit nicely with the three aspects of the meeting (foundations, implementations, certification). Admittedly, the effective analysis part is missing, although I will relate type theory to computable mathematics, for the benefit of those who would like to learn more about type theory and are familiar with computable mathematics.

These slides are available at andrej.com/map2016



# Part I
# Type Theory & Equality Reflection

Andrej Bauer
University of Ljubljana

`andrej.com/map2016`

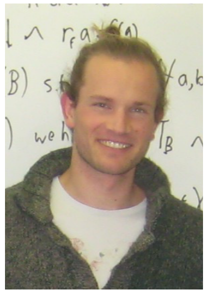
MAP 2016 – Marseille, France

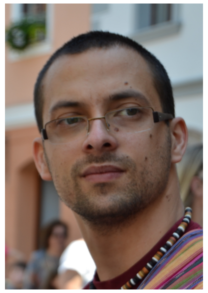
Thank you for the invitation.
The title of this meeting is “Effective Analysis: Foundations, Implementations, Certification”. I will give three lectures, which fit nicely with the three aspects of the meeting (foundations, implementations, certification). Admittedly, the effective analysis part is missing, although I will relate type theory to computable mathematics, for the benefit of those who would like to learn more about type theory and are familiar with computable mathematics.

These slides are available at andrej.com/map2016

## Coworkers

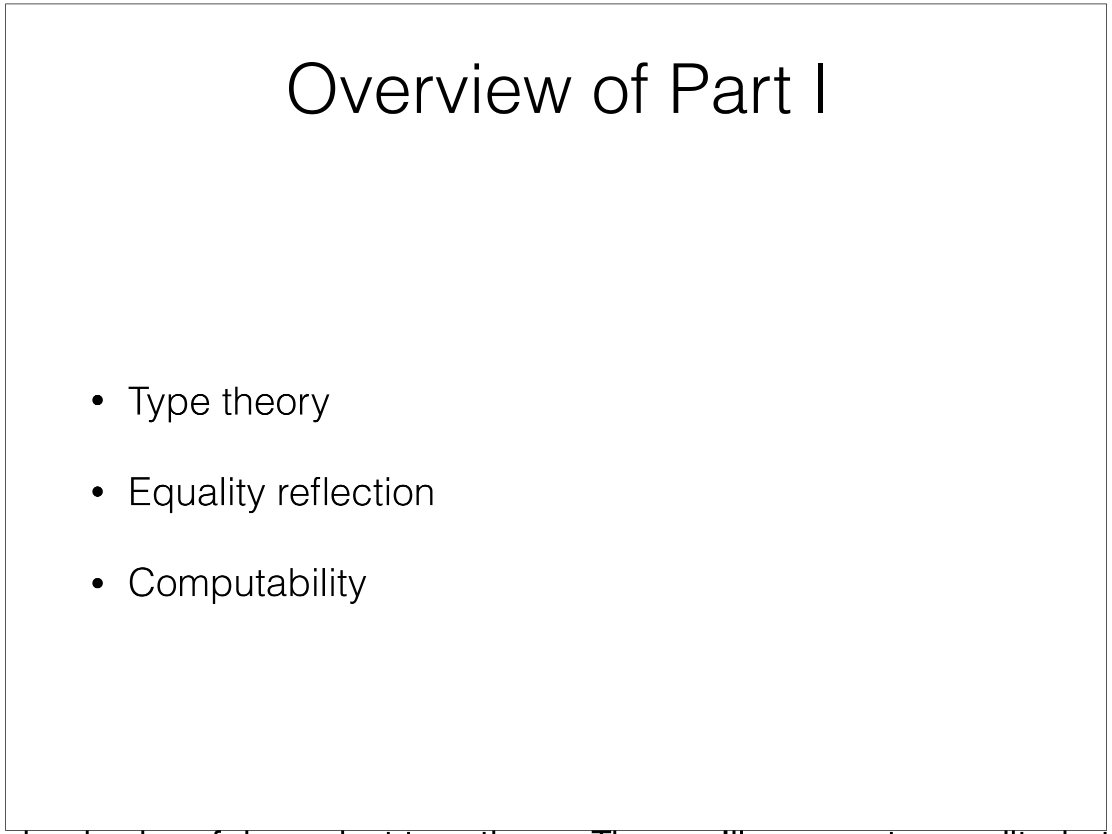
Gaëtan Gilbert


Phillip Haselwarter


Matija Pretnar

Chris Stone

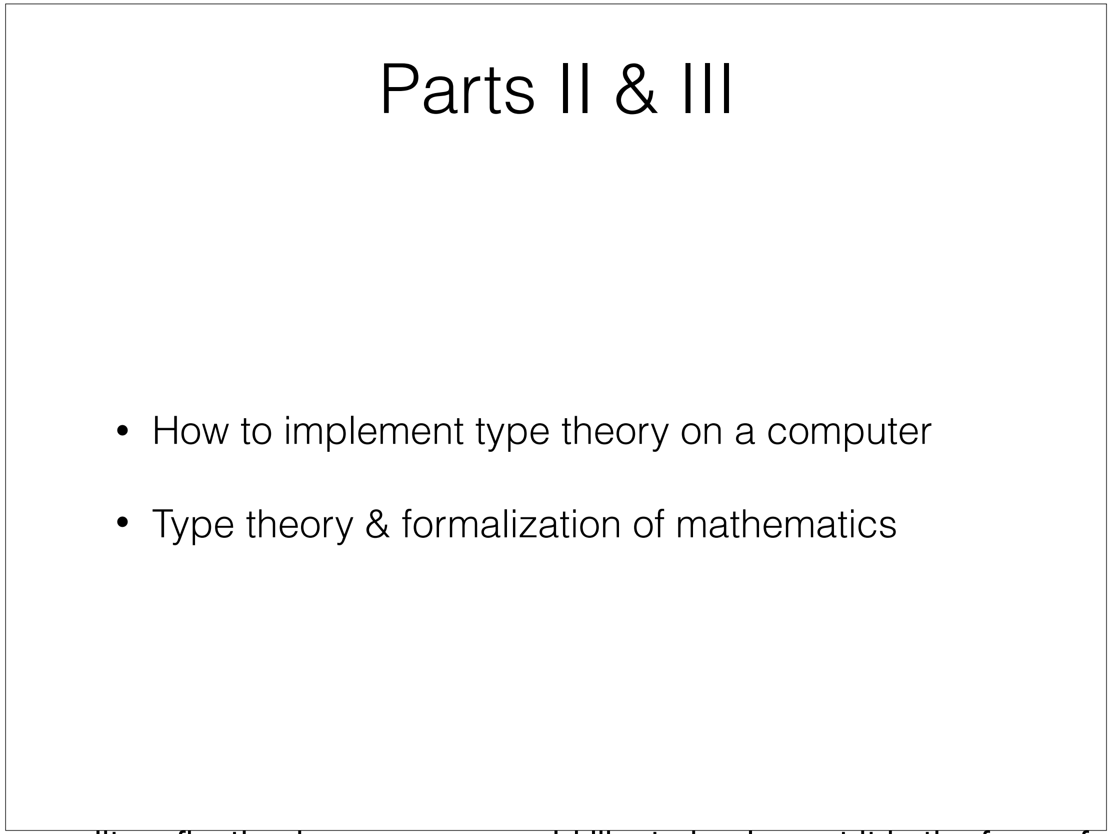
The work I am presenting is done jointly with several people: Gaëtan Gilbert from ENS Lyon, Chris Stone from Harvey Mudd College, USA, Matija Pretnar and Phillip Haselwarter from University of Ljubljana. I like to put up pictures of my coauthors on the slides, so here they are.



## Overview of Part I

- Type theory
- Equality reflection
- Computability

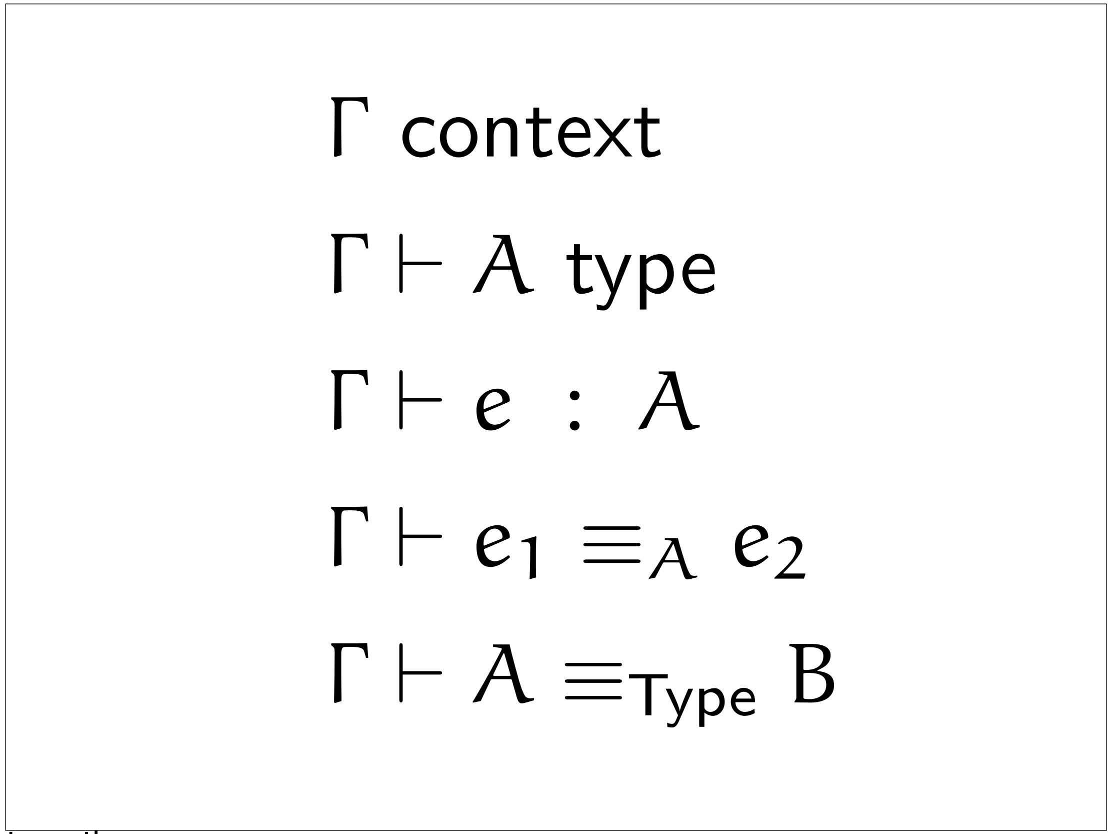
In today's talk we are going to review basics of dependent type theory. Then we'll move onto equality, but unlike in Homotopy Type Theory we'll make equality homotopically and computationally trivial by imposing the so-called **equality reflection rule**. Equality reflection is known among type theorists to cause all sorts of trouble, which we will address, but we shall see that it is compatible with a computational interpretation of type theory. All of today's lecture is a review of known things.



## Parts II & III

- How to implement type theory on a computer
- Type theory & formalization of mathematics

We are studying type theory with equality reflection because we would like to implement it in the form of a proof checker or a proof assistant. This will be the topic of the second lecture, and in the third lecture we will see how the implementation allows us to implement techniques that are used by proof assistants.



$$\Gamma \text{ context}$$

$$\Gamma \vdash A \text{ type}$$

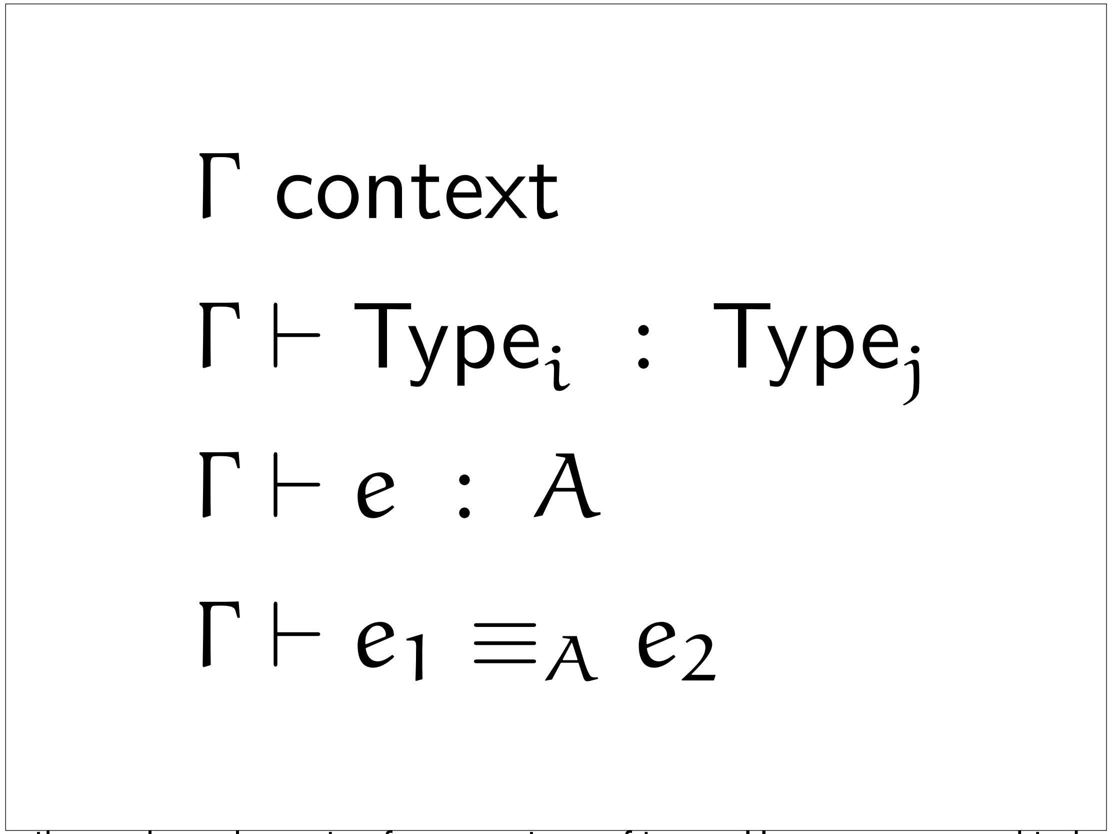
$$\Gamma \vdash e : A$$

$$\Gamma \vdash e_1 \equiv_A e_2$$

$$\Gamma \vdash A \equiv_{\mathsf{Type}} B$$

There are many ways to think of type theory.
A computer-scientists thinks of it as a fancy programming language.
We shall view it from a more mathematical perspective. It is an equational theory. That is, we can construct types and terms, and write down equations between them. And *that is all.* There is no logic, other than equations. There is no set theory. Type theory as a formal system stands on itself. It is possible to bring in logic later *inside* type theory.

But first let us review the parts which constitute type theory. These are called *judgements.* Think of them as the things you can state. Here we have five different judgements. Let me simplify that.



$\Gamma$ context

$\Gamma \vdash \mathsf{Type}_i \ : \ \mathsf{Type}_j$

$\Gamma \vdash e \ : \ A$

$\Gamma \vdash e_1 \equiv_A e_2$

We can treat types as if they were themselves elements of a super-type of types. However, we need to be careful about paradoxes, so we are forced to introduce a whole hierarchy of super-types of types. This is similar to the set-class distinction in set theory.

For the rest of the lectures we are not going to dwell on this point. We are just going to pretend that there is a Type of all types, keeping in mind that there we swept away important technicalities. They can be put back in later.

$$\Gamma \text{ context}$$

$$\Gamma \vdash \mathsf{Type}_i \; : \; \mathsf{Type}_j$$

$$\Gamma \vdash e \; : \; A$$

$$\Gamma \vdash e_1 \equiv_A e_2$$

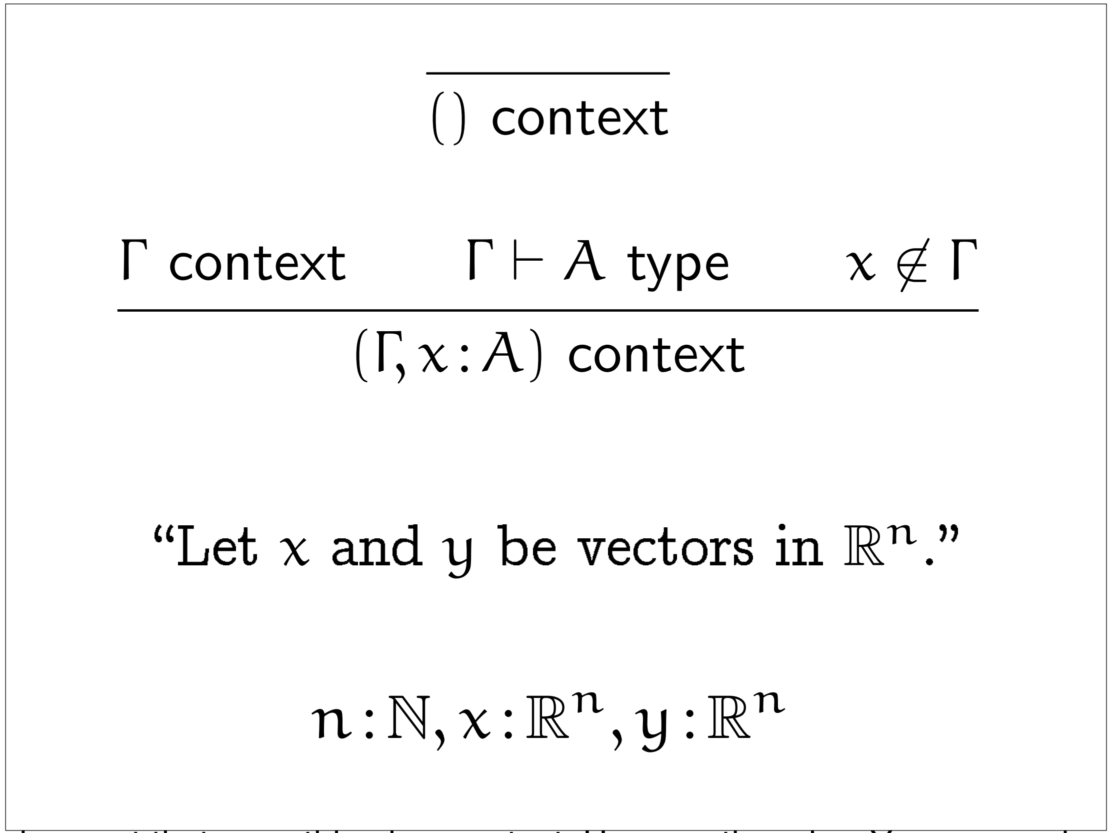
We can treat types as if they were themselves elements of a super-type of types. However, we need to be careful about paradoxes, so we are forced to introduce a whole hierarchy of super-types of types. This is similar to the set-class distinction in set theory.

For the rest of the lectures we are not going to dwell on this point. We are just going to pretend that there is a Type of all types, keeping in mind that there we swept away important technicalities. They can be put back in later.

$$\frac{}{() \text{ context}}$$

$$\frac{\Gamma \text{ context} \qquad \Gamma \vdash A \text{ type} \qquad x \notin \Gamma}{(\Gamma, x : A) \text{ context}}$$

The first thing on our list is the judgement that something is a context. Here are the rules. You can read each “fraction” as saying that if the things above the line hold, then the thing below the line holds.
In informal mathematics contexts are not made explicit, but they are there. For instance, whenever we introduce a new symbol, we’re doing something with the context.



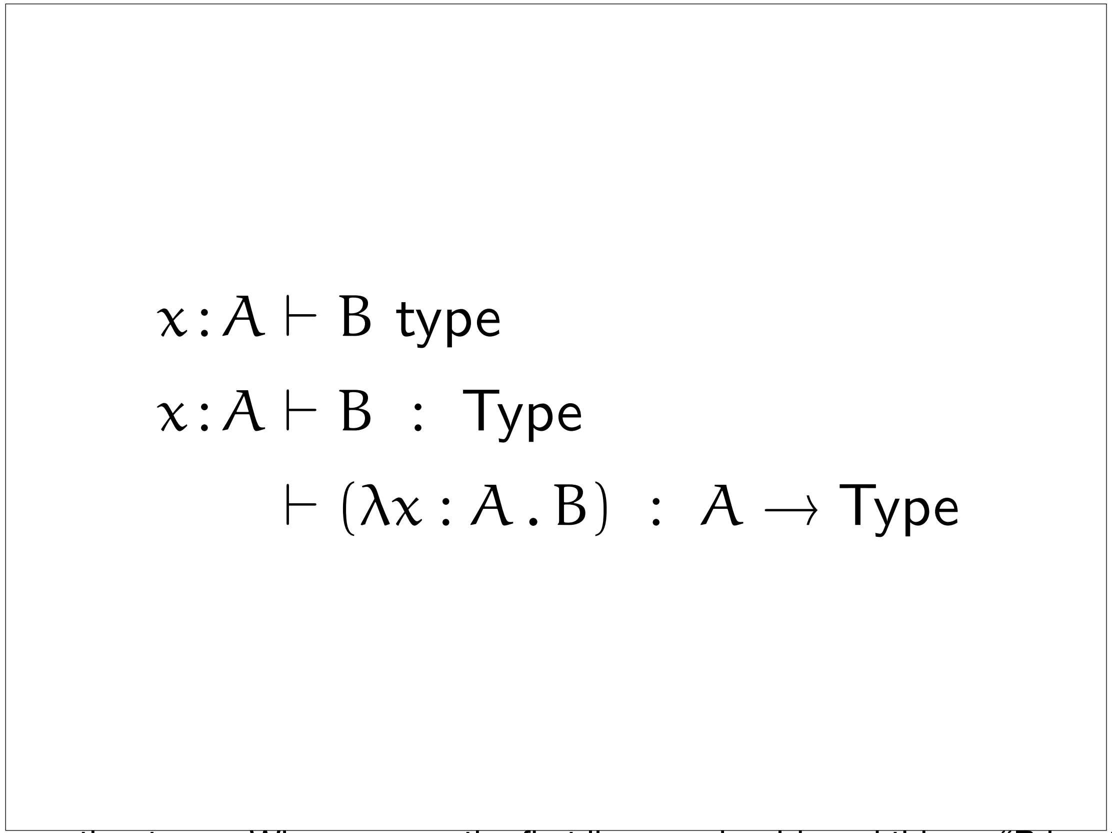
$$\frac{}{()\ \text{context}}$$

$$\frac{\Gamma\ \text{context} \qquad \Gamma \vdash A\ \text{type} \qquad x \notin \Gamma}{(\Gamma, x : A)\ \text{context}}$$

"Let $x$ and $y$ be vectors in $\mathbb{R}^n$."

$$n : \mathbb{N}, x : \mathbb{R}^n, y : \mathbb{R}^n$$

The first thing on our list is the judgement that something is a context. Here are the rules. You can read each "fraction" as saying that if the things above the line hold, then the thing below the line holds.
In informal mathematics contexts are not made explicit, but they are there. For instance, whenever we introduce a new symbol, we're doing something with the context.



$$
\begin{aligned}
x : A &\vdash B \ \text{type} \\
x : A &\vdash B \ : \ \mathsf{Type} \\
&\vdash (\lambda x : A \,.\, B) \ : \ A \to \mathsf{Type}
\end{aligned}
$$

In type theory the types **depend** on other types. When we see the first line we should read this as “B is a type, but which type it is depends on what the value of x is (as well as values of other things in Γ)” Since we assumed there is a type of all types there are two other ways of writing down the fact that B is a type dependent on x from A: (1) B is an element of Type and (2) we have a map from A to types. I have not explained yet what it means to have a map, so you should ignore the third line.

In ordinary mathematics dependencies of types (sets) on elements are extremely common, we just do not notice them because nobody taught us a formalism that actually corresponds to what mathematicians actually write down (set theory is *not* such a formalism).

$$\frac{\Gamma, x{:}A \vdash B \ : \ \mathsf{Type}}{\Gamma \vdash \prod_{x:A} B \ : \ \mathsf{Type}} \qquad \frac{\Gamma, x{:}A \vdash e \ : \ B}{\Gamma \vdash (\lambda x{:}A \,.\, e) \ : \ \prod_{x:A} B}$$

$$\frac{\Gamma \vdash e_1 \ : \ \prod_{x:A} B \qquad \Gamma \vdash e_2 \ : \ A}{\Gamma \vdash e_1\, e_2 \ : \ B[e_2/x]}$$

$$(\lambda x{:}A \,.\, e_1)\, e_2 \equiv_{B[e_2/x]} e_1[e_2/x]$$
$$(\lambda x{:}A \,.\, e\, x) \equiv_{\Pi_{x:A} B} e$$

Our type theory will only have two kinds of types: dependent products and equality types. Here are the rules for dependent products. These are standard, but let us explain them (explanation follows).

The last rule is known as the η-rule. We do not really need it, and we shall see that our implementation allows the user to introduce it if she so whishes.

A special case of dependent product is the non-dependent product A → B, which we get when B does not depend on A. So we have A → B is a short-hand for ∏ (x:A) B where x does not appear in B.

$$\frac{\Gamma, x{:}A \vdash B \ :\ \mathsf{Type}}{\Gamma \vdash \prod_{x:A} B \ :\ \mathsf{Type}} \qquad \frac{\Gamma, x{:}A \vdash e \ :\ B}{\Gamma \vdash (\lambda x{:}A\,.\,e) \ :\ \prod_{x:A} B}$$

$$\frac{\Gamma \vdash e_1 \ :\ \prod_{x:A} B \qquad \Gamma \vdash e_2 \ :\ A}{\Gamma \vdash e_1\, e_2 \ :\ B[e_2/x]}$$

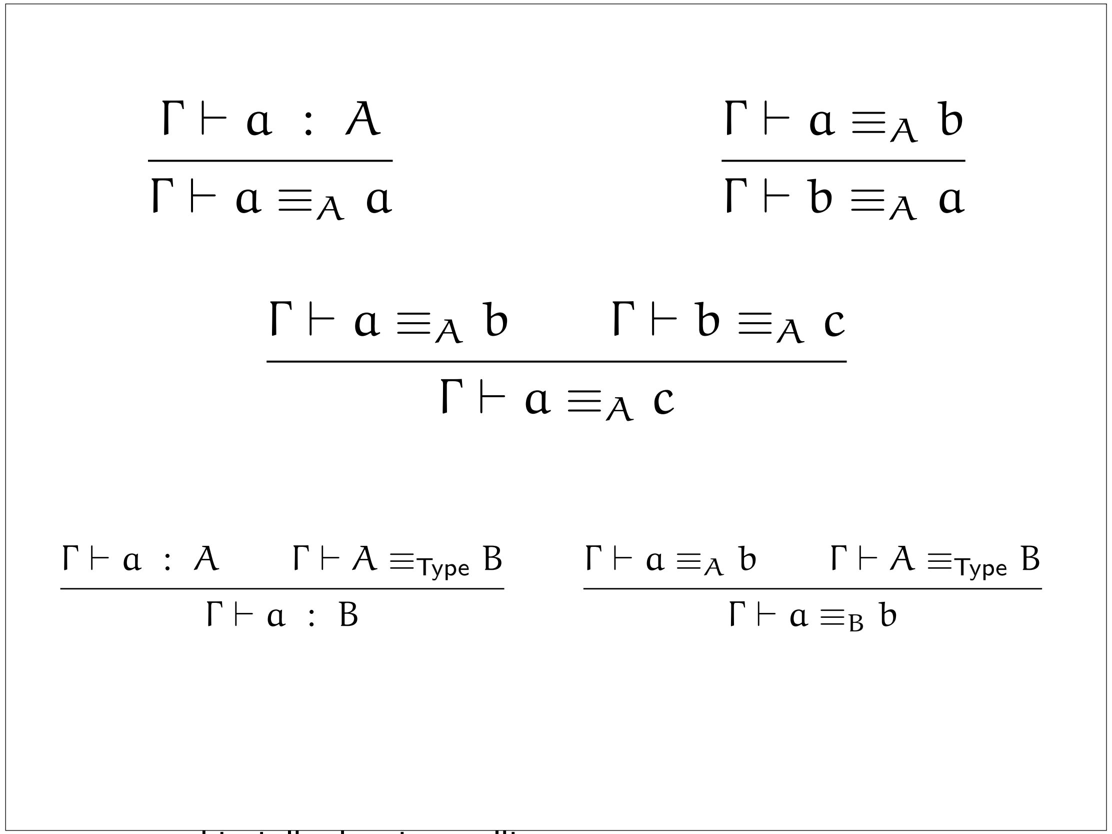
$$(\lambda x{:}A\,.\,e_1)\, e_2 \equiv_{B[e_2/x]} e_1[e_2/x]$$

$$\xcancel{(\lambda x{:}A\,.\,e\,x) \equiv_{\prod_{x:A} B} e}$$

Our type theory will only have two kinds of types: dependent products and equality types. Here are the rules for dependent products. These are standard, but let us explain them (explanation follows).

The last rule is known as the η-rule. We do not really need it, and we shall see that our implementation allows the user to introduce it if she so whishes.

A special case of dependent product is the non-dependent product A → B, which we get when B does not depend on A. So we have A → B is a short-hand for ∏ (x:A) B where x does not appear in B.



$$\frac{\Gamma \vdash a : A}{\Gamma \vdash a \equiv_A a} \qquad \frac{\Gamma \vdash a \equiv_A b}{\Gamma \vdash b \equiv_A a}$$

$$\frac{\Gamma \vdash a \equiv_A b \qquad \Gamma \vdash b \equiv_A c}{\Gamma \vdash a \equiv_A c}$$

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash A \equiv_{\mathsf{Type}} B}{\Gamma \vdash a : B} \qquad \frac{\Gamma \vdash a \equiv_A b \qquad \Gamma \vdash A \equiv_{\mathsf{Type}} B}{\Gamma \vdash a \equiv_B b}$$

Before we can explain equality types we need to talk about equality.
It is reflexive, symmetric and transitive. Notice that we are not allowed to compare two things unless they have the same type.
The last two rules are known as *conversion rules.*

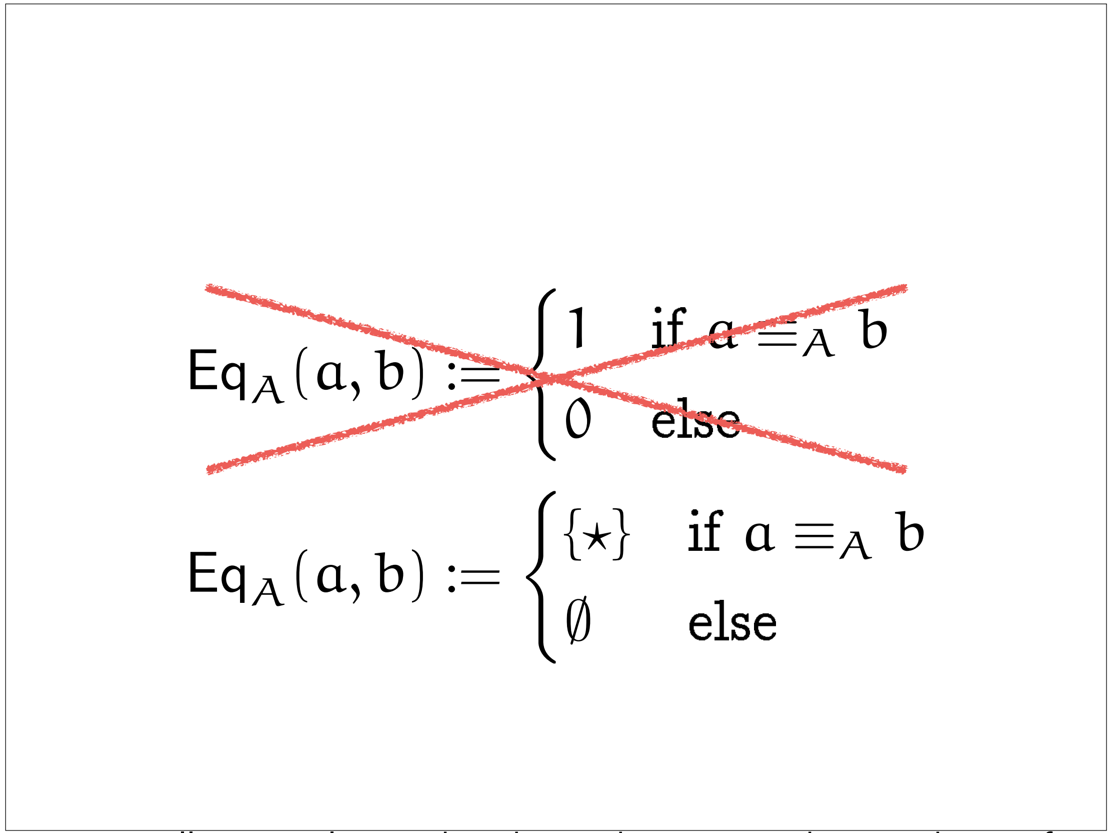
$$\mathsf{Eq}_A(a, b) := \begin{cases} 1 & \text{if } a \equiv_A b \\ 0 & \text{else} \end{cases}$$

$$\mathsf{Eq}_A(a, b) := \begin{cases} \{\star\} & \text{if } a \equiv_A b \\ \emptyset & \text{else} \end{cases}$$

The other type former is equality. Let me call it “equality” rather than “identity” – to keep it distinct from the usual identity type in Martin-Löf intensional type theory.

We would like a type which depends on two elements and tells us whether the two elements are equal. We might try something like a characteristic function, but this is wrong because it is more like a map from A × A → bool, which is not what we want (although this is how you can do equality in a topos or in Church’s simple type theory, which is used by the HOL proof assistant).

A better try is to have Eq(a,b) equal to a singleton or the empty set, depending on whether a and b are equal. But we should not use sets. There are no sets.



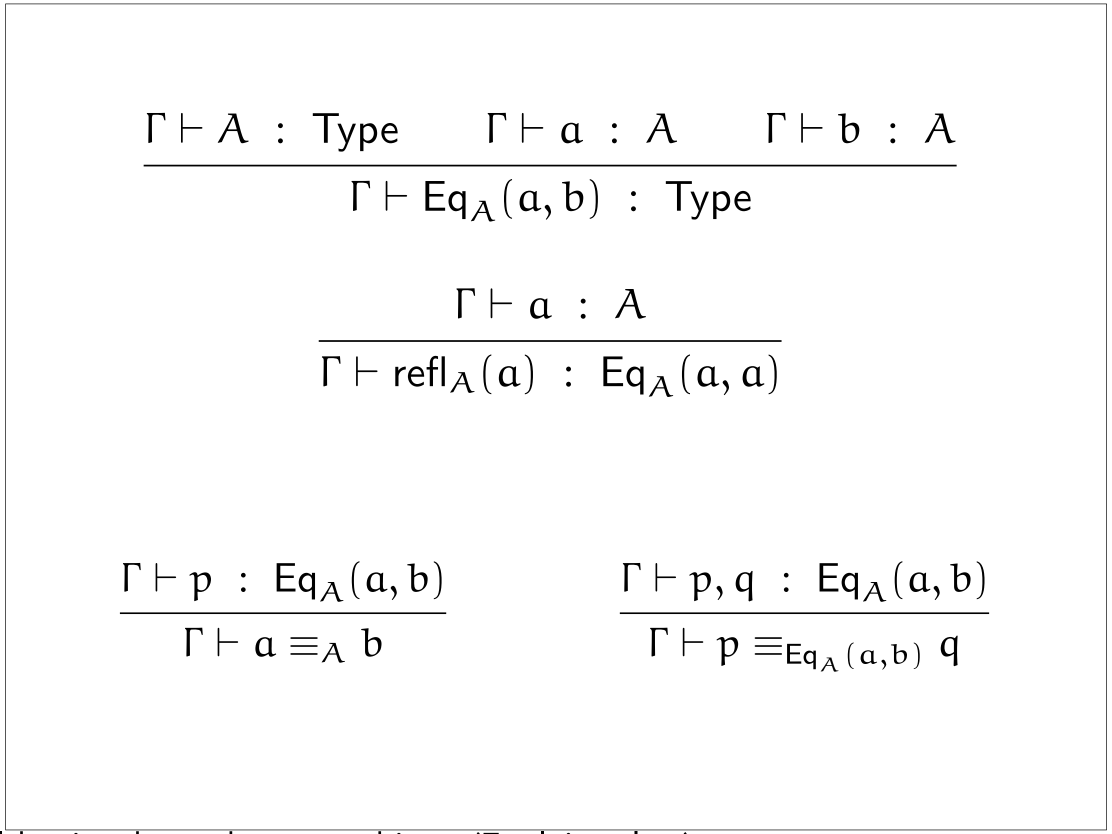
$$\cancel{\mathsf{Eq}_A(a,b) := \begin{cases} 1 & \text{if } a \equiv_A b \\ 0 & \text{else} \end{cases}}$$

$$\mathsf{Eq}_A(a,b) := \begin{cases} \{\star\} & \text{if } a \equiv_A b \\ \emptyset & \text{else} \end{cases}$$

The other type former is equality. Let me call it “equality” rather than “identity” – to keep it distinct from the usual identity type in Martin-Löf intensional type theory.

We would like a type which depends on two elements and tells us whether the two elements are equal. We might try something like a characteristic function, but this is wrong because it is more like a map from A × A → bool, which is not what we want (although this is how you can do equality in a topos or in Church’s simple type theory, which is used by the HOL proof assistant).

A better try is to have Eq(a,b) equal to a singleton or the empty set, depending on whether a and b are equal. But we should not use sets. There are no sets.



$$\frac{\Gamma \vdash A \;:\; \mathsf{Type} \qquad \Gamma \vdash a \;:\; A \qquad \Gamma \vdash b \;:\; A}{\Gamma \vdash \mathsf{Eq}_A(a,b) \;:\; \mathsf{Type}}$$

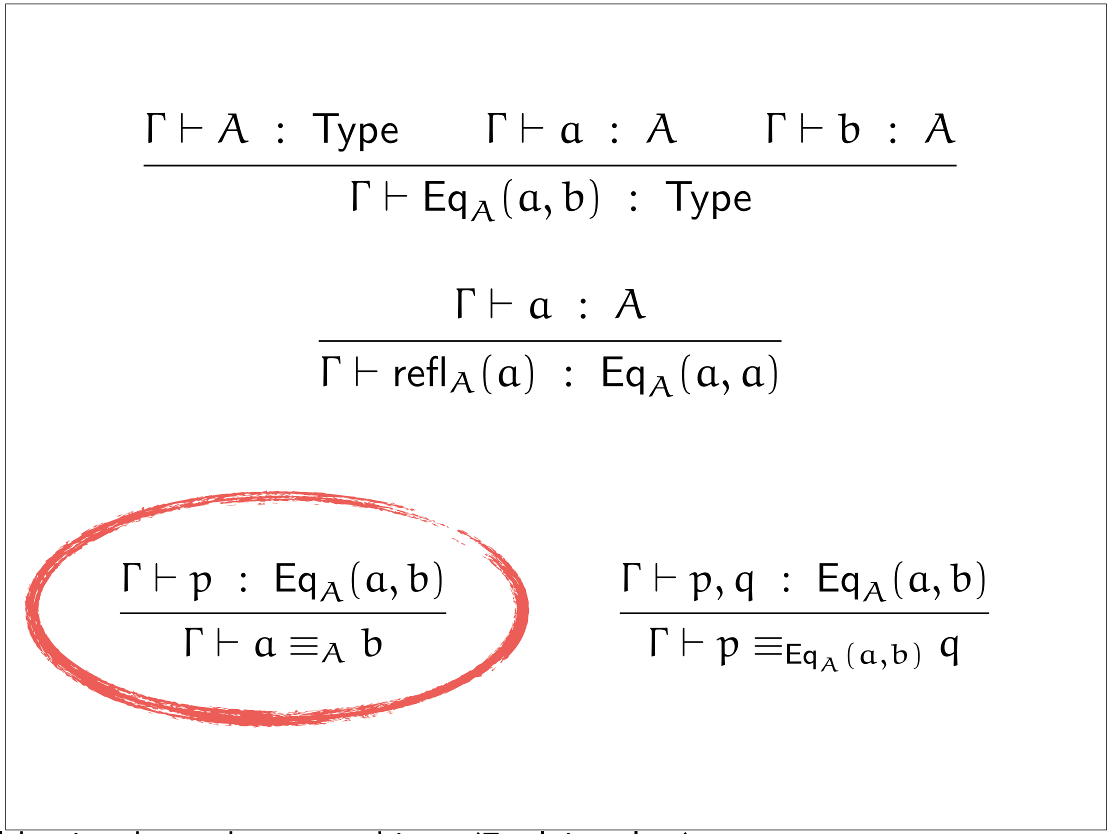
$$\frac{\Gamma \vdash a \;:\; A}{\Gamma \vdash \mathsf{refl}_A(a) \;:\; \mathsf{Eq}_A(a,a)}$$

$$\frac{\Gamma \vdash p \;:\; \mathsf{Eq}_A(a,b)}{\Gamma \vdash a \equiv_A b} \qquad \frac{\Gamma \vdash p, q \;:\; \mathsf{Eq}_A(a,b)}{\Gamma \vdash p \equiv_{\mathsf{Eq}_A(a,b)} q}$$

Here is how a type theorist would write down the same thing. (Explain rules.)

The rule on the bottom left is known as equality reflection. It tells us that the equality type Eq really corresponds to equality. The other direction, if a and b are equal then Eq(a,b) has an element, follows from conversion.

The rule at the bottom right says that an equality type has at most one element. The way this is stated with an equation is that every p of type Eq(a,b) is equal to refl(a). Exercise: why does refl(a) have type Eq(a,b)?



$$\frac{\Gamma \vdash A \ : \ \mathsf{Type} \qquad \Gamma \vdash a \ : \ A \qquad \Gamma \vdash b \ : \ A}{\Gamma \vdash \mathsf{Eq}_A(a, b) \ : \ \mathsf{Type}}$$

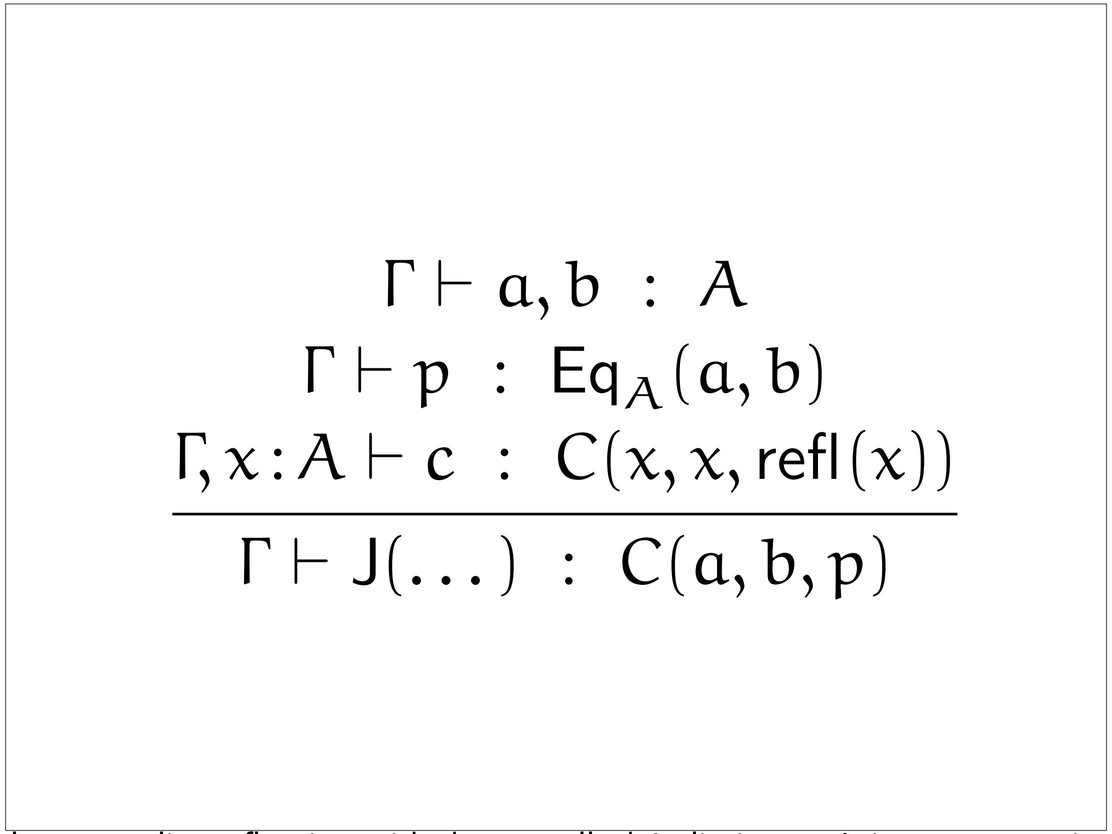
$$\frac{\Gamma \vdash a \ : \ A}{\Gamma \vdash \mathsf{refl}_A(a) \ : \ \mathsf{Eq}_A(a, a)}$$

$$\frac{\Gamma \vdash p \ : \ \mathsf{Eq}_A(a, b)}{\Gamma \vdash a \equiv_A b} \qquad\qquad \frac{\Gamma \vdash p, q \ : \ \mathsf{Eq}_A(a, b)}{\Gamma \vdash p \equiv_{\mathsf{Eq}_A(a,b)} q}$$

Here is how a type theorist would write down the same thing. (Explain rules.)

The rule on the bottom left is known as equality reflection. It tells us that the equality type Eq really corresponds to equality. The other direction, if a and b are equal then Eq(a,b) has an element, follows from conversion.

The rule at the bottom right says that an equality type has at most one element. The way this is stated with an equation is that every p of type Eq(a,b) is equal to refl(a). Exercise: why does refl(a) have type Eq(a,b)?



$$\frac{\begin{array}{c}\Gamma \vdash a, b \ :\ A \\ \Gamma \vdash p \ :\ \mathsf{Eq}_A(a, b) \\ \Gamma, x : A \vdash c \ :\ C(x, x, \mathsf{refl}(x))\end{array}}{\Gamma \vdash \mathsf{J}(\dots) \ :\ C(a, b, p)}$$

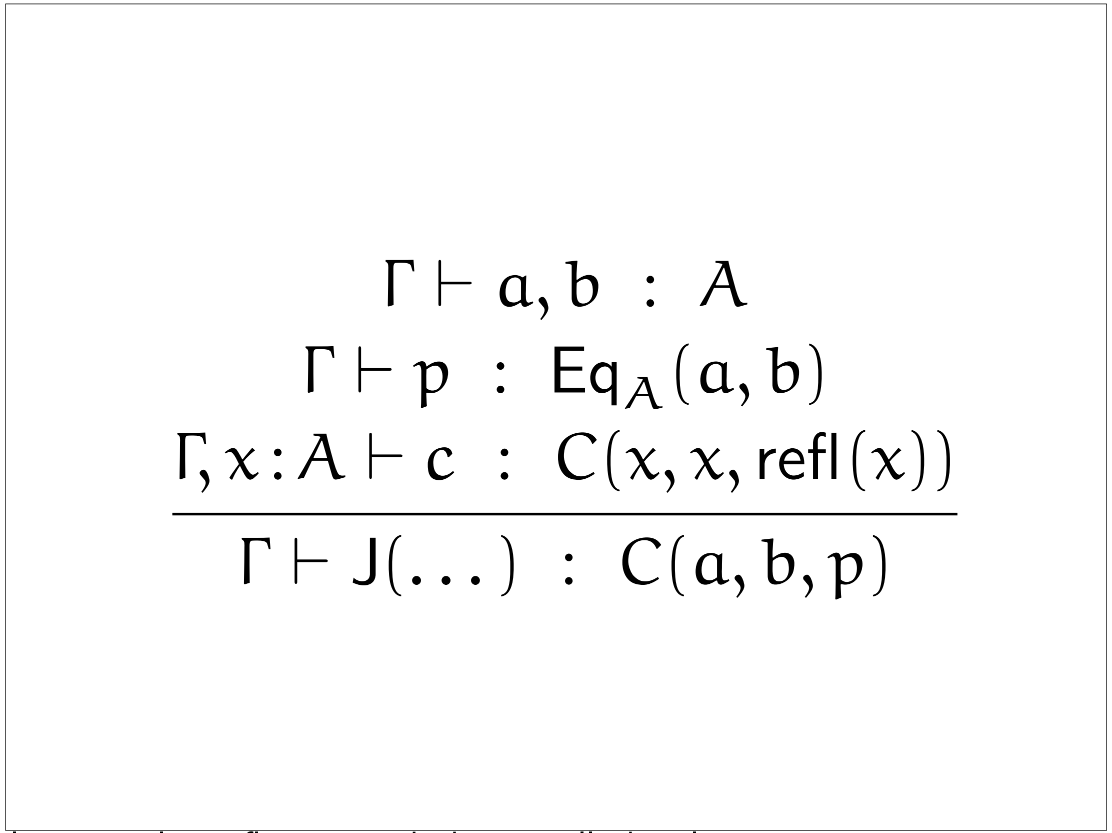
In *intensional type theory* we replace equality reflection with the so-called J-eliminator. It is a construction witnessing the fact that equality is the least reflexive relation (explain).

We can get J from equality reflection and its equality rule, but it does not do anything interesting. In Homotopy Type Theory they (we) wrote a hole book about it.

$$\frac{\begin{array}{c}\Gamma \vdash a, b \ : \ A \\ \Gamma \vdash p \ : \ \mathsf{Eq}_A(a, b) \\ \Gamma, x : A \vdash c \ : \ C(x, x, \mathsf{refl}(x))\end{array}}{\Gamma \vdash \mathsf{J}(\dots) \ : \ C(a, b, p)}$$

In *intensional type theory* we replace equality reflection with the so-called J-eliminator. It is a construction witnessing the fact that equality is the least reflexive relation (explain).

We can get J from equality reflection and its equality rule, but it does not do anything interesting. In Homotopy Type Theory they (we) wrote a hole book about it.



$$\frac{\begin{array}{c}\Gamma \vdash a, b \ : \ A \\ \Gamma \vdash p \ : \ \mathsf{Eq}_A(a, b) \\ \Gamma, x : A \vdash c \ : \ C(x, x, \mathsf{refl}(x))\end{array}}{\Gamma \vdash \mathsf{J}(\dots) \ : \ C(a, b, p)}$$

In *intensional type theory* we replace equality reflection with the so-called J-eliminator. It is a construction witnessing the fact that equality is the least reflexive relation (explain).

We can get J from equality reflection and its equality rule, but it does not do anything interesting. In Homotopy Type Theory they (we) wrote a hole book about it.

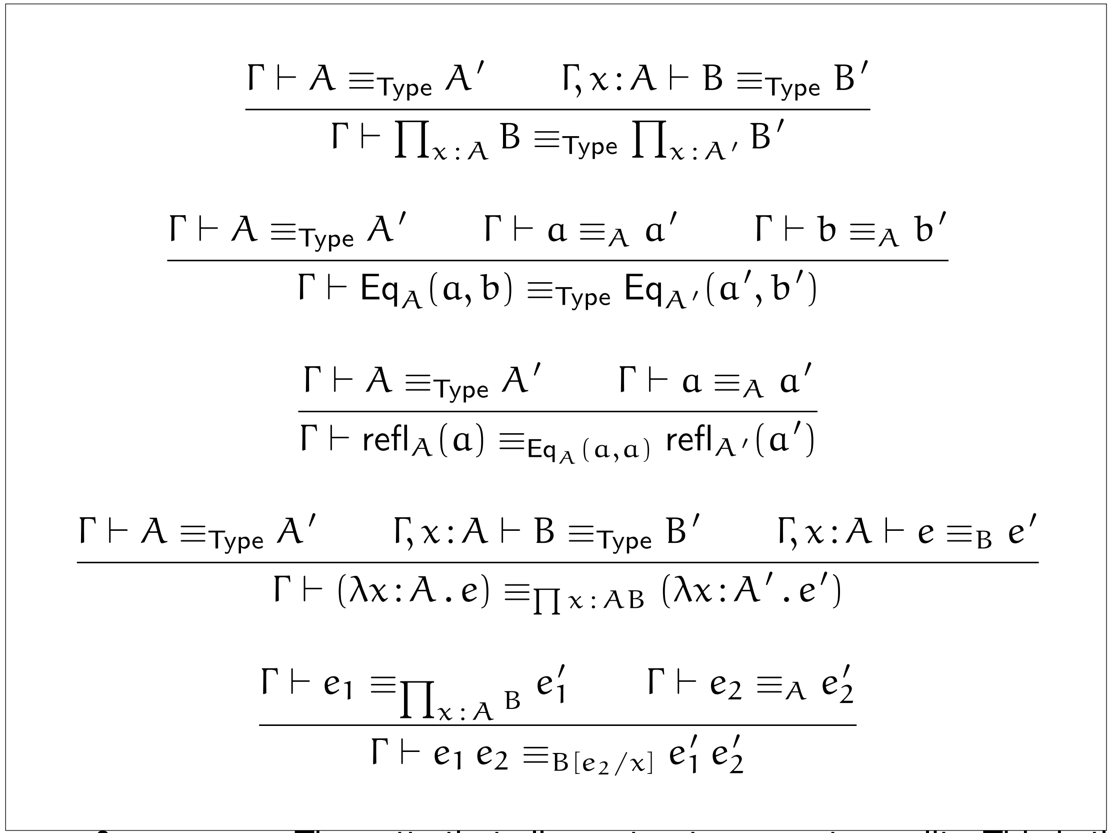
$$\frac{\Gamma \vdash a, b \ : \ A \qquad \Gamma \vdash p \ : \ \mathsf{Eq}_A(a, b) \qquad \Gamma, x : A \vdash c \ : \ C(x, x, \mathsf{refl}(x))}{\Gamma \vdash \mathsf{J}(\ldots) \ : \ C(a, b, p)}$$

$\nwarrow$

just use $c[a/x]$ for $J(\ldots)$

In *intensional type theory* we replace equality reflection with the so-called J-eliminator. It is a construction witnessing the fact that equality is the least reflexive relation (explain).

We can get J from equality reflection and its equality rule, but it does not do anything interesting. In Homotopy Type Theory they (we) wrote a hole book about it.



$$\frac{\Gamma \vdash A \equiv_{\mathsf{Type}} A' \qquad \Gamma, x{:}A \vdash B \equiv_{\mathsf{Type}} B'}{\Gamma \vdash \prod_{x:A} B \equiv_{\mathsf{Type}} \prod_{x:A'} B'}$$

$$\frac{\Gamma \vdash A \equiv_{\mathsf{Type}} A' \qquad \Gamma \vdash a \equiv_{A} a' \qquad \Gamma \vdash b \equiv_{A} b'}{\Gamma \vdash \mathsf{Eq}_A(a, b) \equiv_{\mathsf{Type}} \mathsf{Eq}_{A'}(a', b')}$$

$$\frac{\Gamma \vdash A \equiv_{\mathsf{Type}} A' \qquad \Gamma \vdash a \equiv_{A} a'}{\Gamma \vdash \mathsf{refl}_A(a) \equiv_{\mathsf{Eq}_A(a,a)} \mathsf{refl}_{A'}(a')}$$

$$\frac{\Gamma \vdash A \equiv_{\mathsf{Type}} A' \qquad \Gamma, x{:}A \vdash B \equiv_{\mathsf{Type}} B' \qquad \Gamma, x{:}A \vdash e \equiv_{B} e'}{\Gamma \vdash (\lambda x{:}A\,.\,e) \equiv_{\prod x:A B} (\lambda x{:}A'\,.\,e')}$$

$$\frac{\Gamma \vdash e_1 \equiv_{\prod_{x:A} B} e_1' \qquad \Gamma \vdash e_2 \equiv_{A} e_2'}{\Gamma \vdash e_1\, e_2 \equiv_{B[e_2/x]} e_1'\, e_2'}$$

Congruence rules are boring but easy & necessary. They stte that all constructs respect equality. This is the sort of detail that seem unecessary until we try to prove that we correctly implemented a proof assistants.

$$\frac{\Gamma \vdash A \equiv_{\mathsf{Type}} A' \qquad \Gamma, x{:}A \vdash B \equiv_{\mathsf{Type}} B'}{\Gamma \vdash \prod_{x:A} B \equiv_{\mathsf{Type}} \prod_{x:A'} B'}$$

$$\frac{\Gamma \vdash A \equiv_{\mathsf{Type}} A' \qquad \Gamma \vdash a \equiv_A a' \qquad \Gamma \vdash b \equiv_A b'}{\Gamma \vdash \mathsf{Eq}_A(a, b) \equiv_{\mathsf{Type}} \mathsf{Eq}_{A'}(a', b')}$$

$$\frac{\Gamma \vdash A \equiv_{\mathsf{Type}} A' \qquad \Gamma \vdash a \equiv_A a'}{\Gamma \vdash \mathsf{refl}_A(a) \equiv_{\mathsf{Eq}_A(a,a)} \mathsf{refl}_{A'}(a')}$$

$$\frac{\Gamma \vdash A \equiv_{\mathsf{Type}} A' \qquad \Gamma, x{:}A \vdash B \equiv_{\mathsf{Type}} B' \qquad \Gamma, x{:}A \vdash e \equiv_B e'}{\Gamma \vdash (\lambda x{:}A \,.\, e) \equiv_{\prod x:A B} (\lambda x{:}A' \,.\, e')}$$

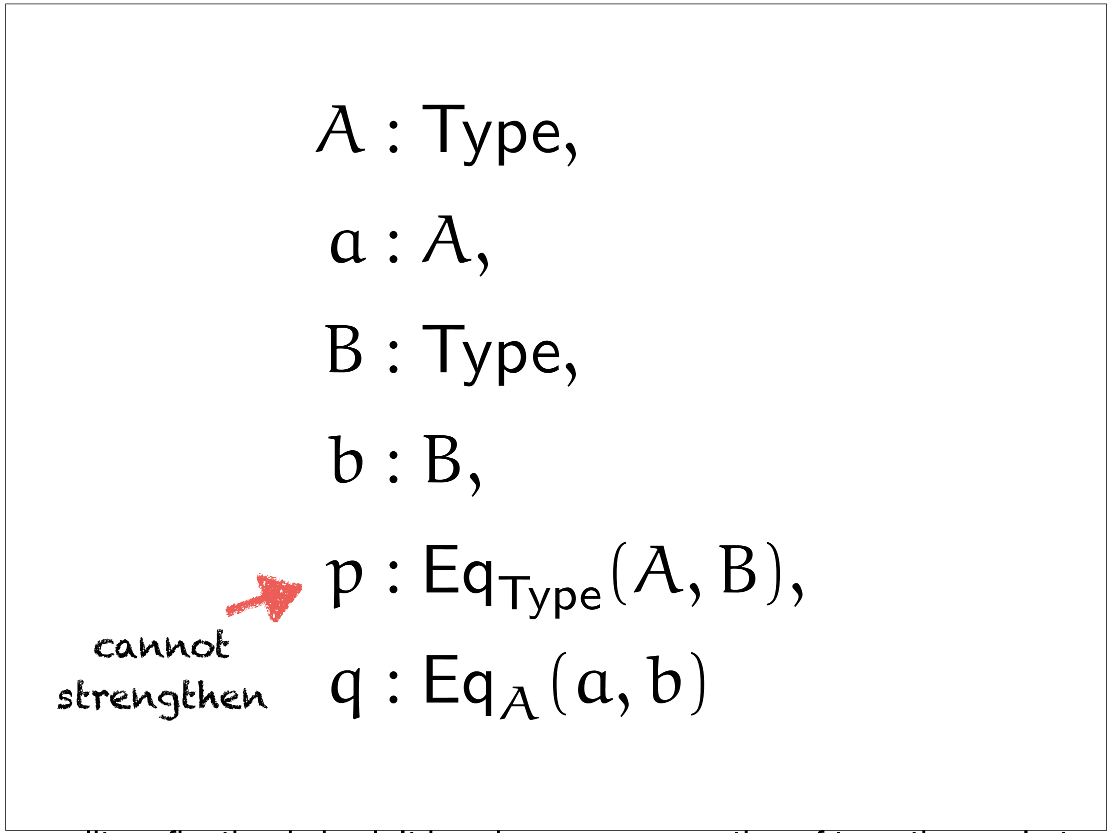
$$\frac{\Gamma \vdash e_1 \equiv_{\prod_{x:A} B} e_1' \qquad \Gamma \vdash e_2 \equiv_A e_2'}{\Gamma \vdash e_1\, e_2 \equiv_{B[e_2/x]} e_1'\, e_2'}$$

Congruence rules are boring but easy & necessary. They stte that all constructs respect equality. This is the sort of detail that seem unecessary until we try to prove that we correctly implemented a proof assistants.

$$
\begin{aligned}
&A : \mathsf{Type},\\
&a : A,\\
&B : \mathsf{Type},\\
&b : B,\\
&p : \mathsf{Eq}_{\mathsf{Type}}(A, B),\\
&q : \mathsf{Eq}_{A}(a, b)
\end{aligned}
$$

It is known to type theorists that equality reflection is bad. It breaks many properties of type theory. Let us see.

Consider the following context, written vertically. We have two types A and B with elements a and b.
The type of q makes sense because, by reflection, it says that A and B are equal. We cannot remove p from the context, even though it is not mentioned anywhere, so **strengthening is not valid**. Similarly, a naive formulation of exchange (in terms of p and q not appearing in the types) would be invalid.

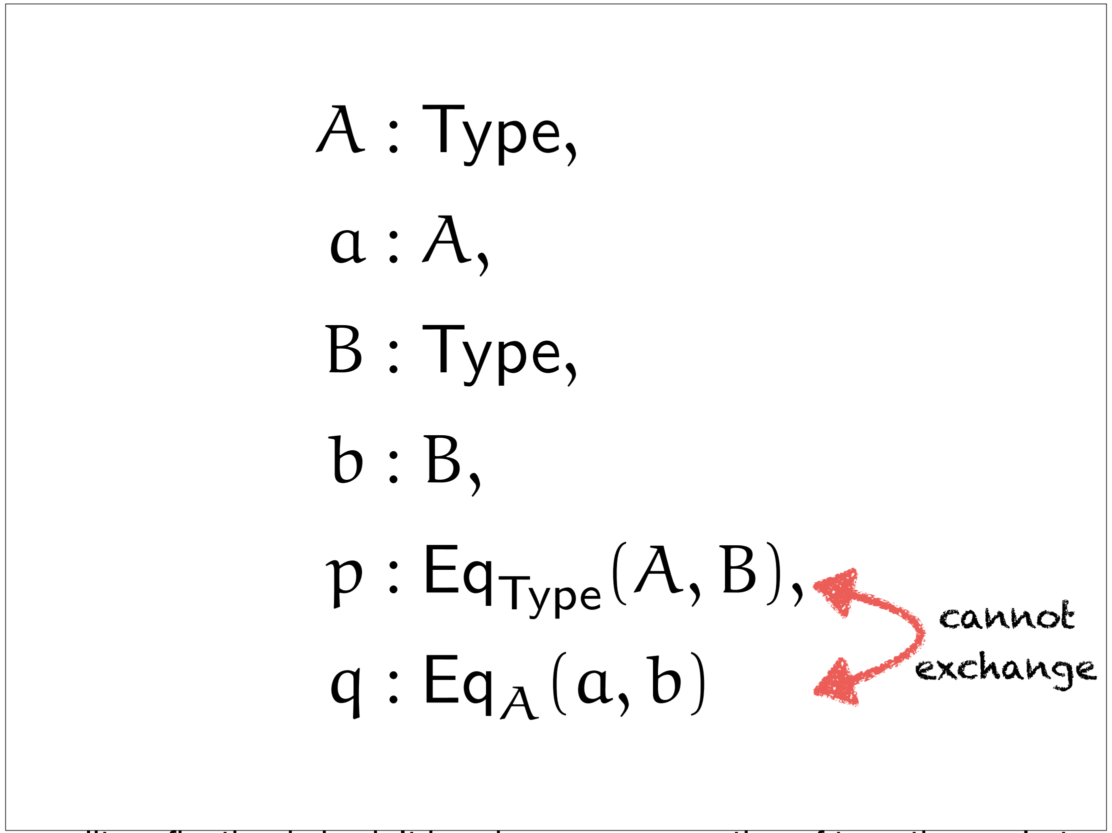


$$
\begin{aligned}
A &: \mathsf{Type}, \\
a &: A, \\
B &: \mathsf{Type}, \\
b &: B, \\
p &: \mathsf{Eq}_{\mathsf{Type}}(A, B), \\
q &: \mathsf{Eq}_A(a, b)
\end{aligned}
$$

cannot strengthen

It is known to type theorists that equality reflection is bad. It breaks many properties of type theory. Let us see.

Consider the following context, written vertically. We have two types A and B with elements a and b.
The type of q makes sense because, by reflection, it says that A and B are equal. We cannot remove p from the context, even though it is not mentioned anywhere, so **strengthening is not valid**. Similarly, a naive formulation of exchange (in terms of p and q not appearing in the types) would be invalid.



$$\begin{array}{l} A : \mathrm{Type}, \\ a : A, \\ B : \mathrm{Type}, \\ b : B, \\ p : \mathrm{Eq}_{\mathrm{Type}}(A, B), \\ q : \mathrm{Eq}_{A}(a, b) \end{array}$$

cannot exchange

It is known to type theorists that equality reflection is bad. It breaks many properties of type theory. Let us see.

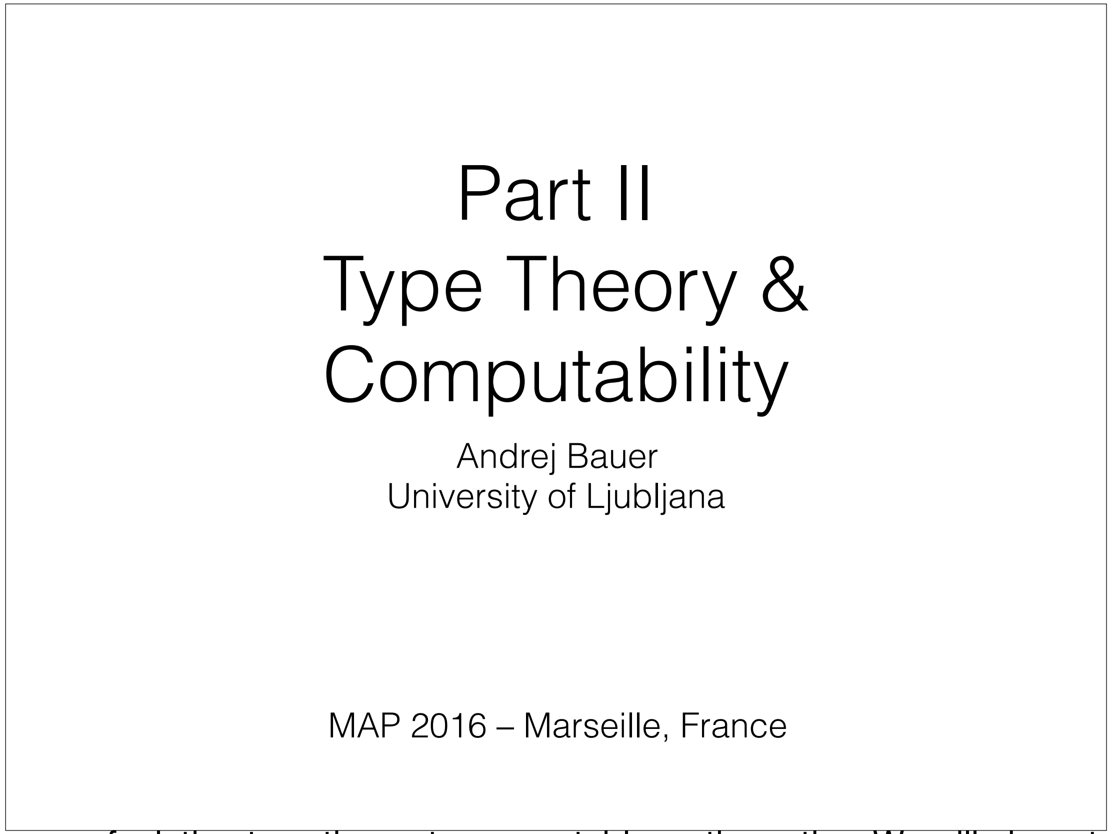
Consider the following context, written vertically. We have two types A and B with elements a and b.
The type of q makes sense because, by reflection, it says that A and B are equal. We cannot remove p from the context, even though it is not mentioned anywhere, so **strengthening is not valid**. Similarly, a naive formulation of exchange (in terms of p and q not appearing in the types) would be invalid.

$$
\begin{aligned}
&p : \mathsf{Eq}_{\mathsf{Type}}(\mathsf{nat} \to \mathsf{nat}, \mathsf{nat} \to \mathsf{bool}),\\
&m : \mathsf{nat}\\
&\vdash\\
&m \equiv_{\mathsf{bool}} (\lambda x : \mathsf{nat} \,.\, x) m \ : \ \mathsf{bool}
\end{aligned}
$$

It gets worse. The rules we have written down are actually not acceptable because they allow us to write down very strange things. For instance, if we assume p : Eq(nat → nat, nat → bool) then we get an injective map nat → bool, which leads to a contradiction! (I had a proof of this in September in Paris, but now I forgot it, so it is an exercise to be solved by Wednesday.)

The assumption p : Eq(nat → nat, nat → bool) is *valid* in some models of type theory, for instance in the skeleton of the category of sets (i.e., the topos of cardinal numbers and all functions between them). So we either have to give up the expected notion of model or fix the type theory. Let us do the latter.

## End of Part I

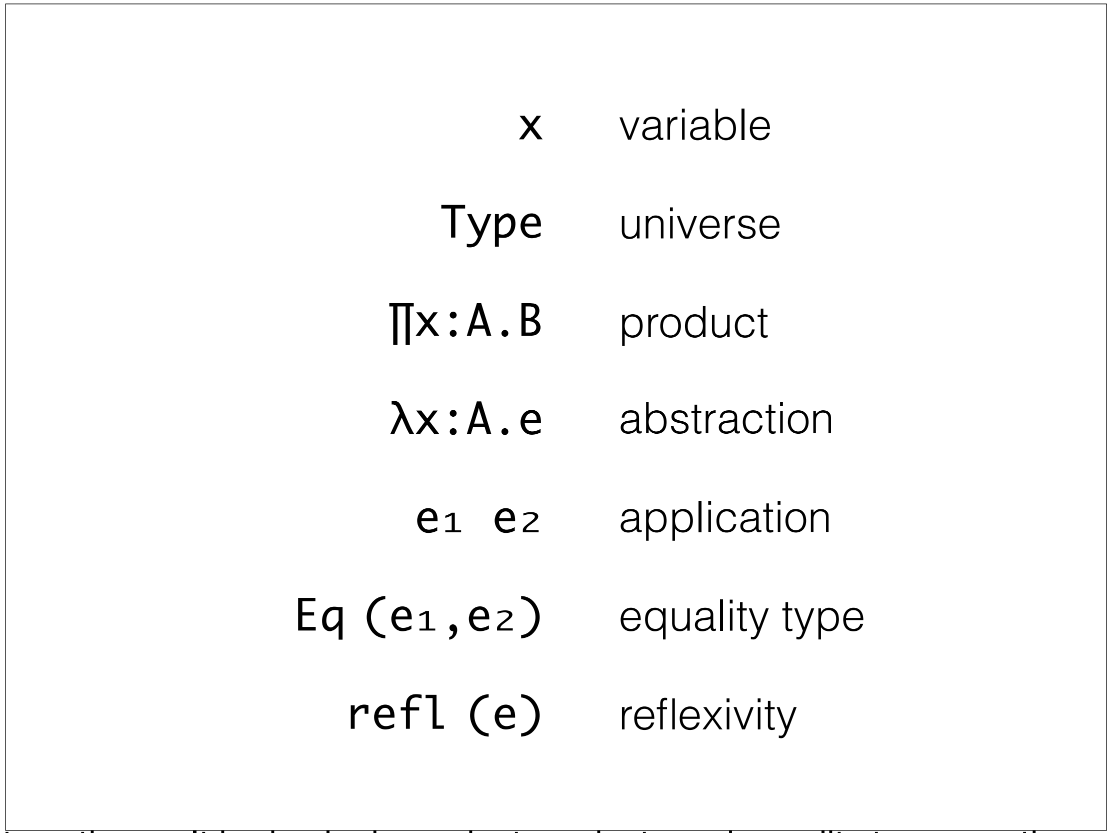


# Part II Type Theory & Computability

Andrej Bauer
University of Ljubljana

MAP 2016 – Marseille, France

Today I would like to explain one way of relating type theory to computable mathematics. We will give a translation of type theory (logicians would call it a *model*) in which the types will be represented sets. But before that let us take care of some loose end from yesterday.



| | |
|---|---|
| `x` | variable |
| `Type` | universe |
| `∏x:A.B` | product |
| `λx:A.e` | abstraction |
| $e_1\ e_2$ | application |
| $\mathtt{Eq}\ (e_1, e_2)$ | equality type |
| `refl (e)` | reflexivity |

In the first lecture we discussed type theory. It had only dependent products and equality types, so the syntax was pretty simple.

$$\frac{\Gamma \text{ context} \qquad \Gamma \vdash A : \mathsf{Type} \qquad \Gamma \vdash a : A}{\Gamma \vdash \mathsf{refl}_A(a) \ : \ \mathsf{Eq}_A(a, a)}$$

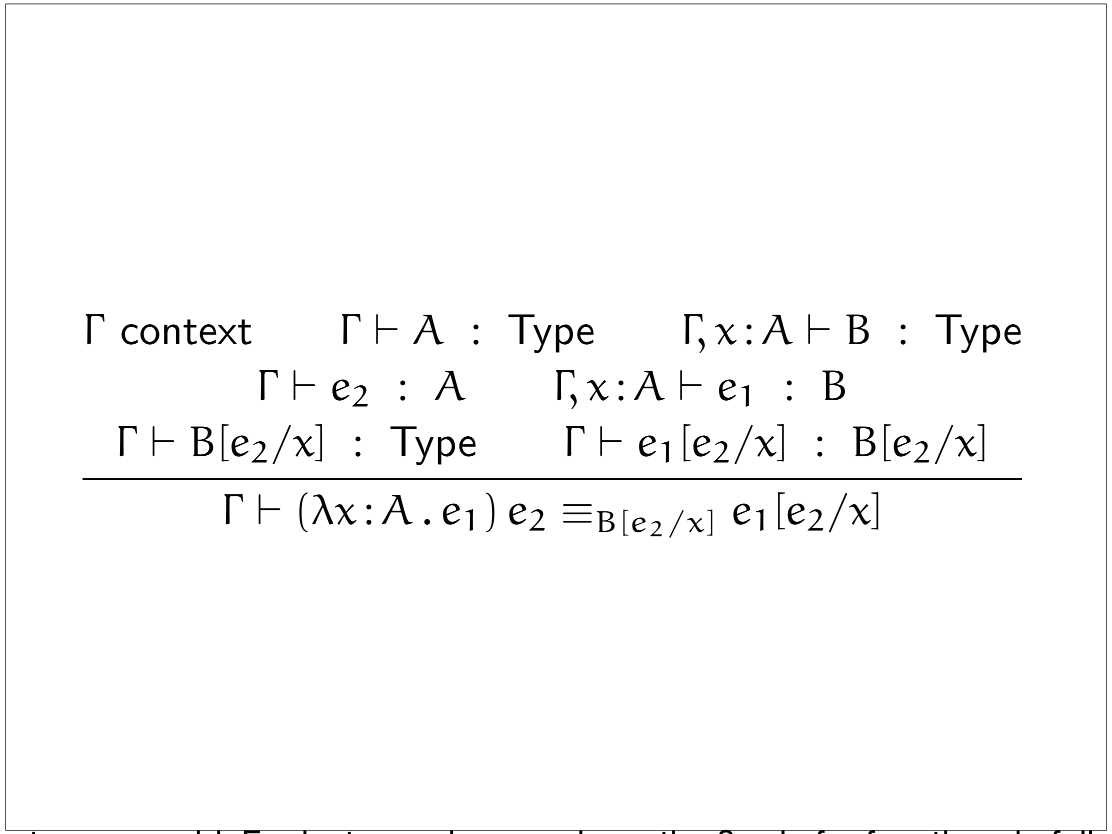
Yesterday I pointed out (on the blackboard) that one really should be quite paranoid about putting in sufficiently many premises, to guarantee everything was well-formed. For instance, the reflexivity rule can be made more paranoid.

Depending on how things are set up, we may not actually have to put in all these extra assumptions. For instance, perhaps we can prove a meta-theorem saying that whenever $\Gamma \vdash a : A$ then $\Gamma$ is a context and A is a type. But it is better to be paranoid first and then notice later which premises are unnecessary.

$$\frac{\Gamma \text{ context} \qquad \Gamma \vdash A : \mathsf{Type} \qquad \Gamma \vdash a : A}{\Gamma \vdash \mathsf{refl}_A(a) : \mathsf{Eq}_A(a, a)}$$

Yesterday I pointed out (on the blackboard) that one really should be quite paranoid about putting in sufficiently many premises, to guarantee everything was well-formed. For instance, the reflexivity rule can be made more paranoid.

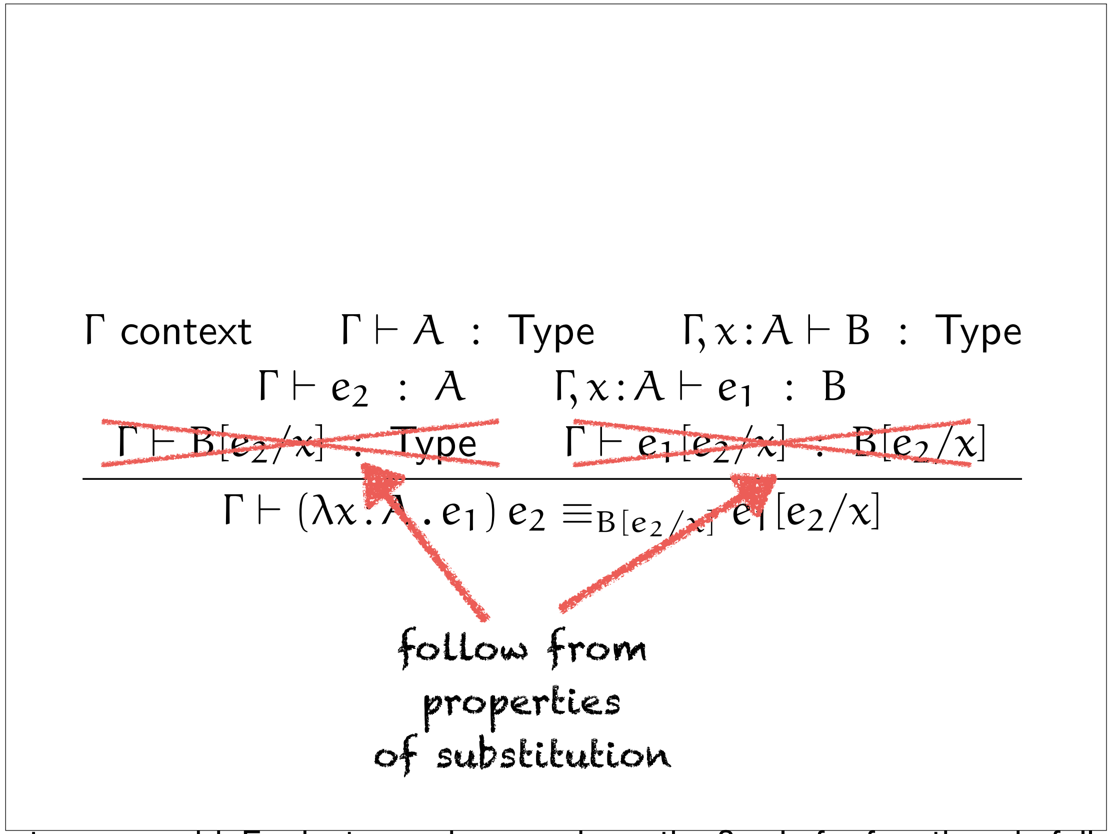
Depending on how things are set up, we may not actually have to put in all these extra assumptions. For instance, perhaps we can prove a meta-theorem saying that whenever $\Gamma \vdash a : A$ then $\Gamma$ is a context and A is a type. But it is better to be paranoid first and then notice later which premises are unnecessary.



$$\frac{\begin{array}{c}\Gamma\ \text{context} \qquad \Gamma \vdash A \ :\ \text{Type} \qquad \Gamma, x : A \vdash B \ :\ \text{Type} \\ \Gamma \vdash e_2 \ :\ A \qquad \Gamma, x : A \vdash e_1 \ :\ B \\ \Gamma \vdash B[e_2/x] \ :\ \text{Type} \qquad \Gamma \vdash e_1[e_2/x] \ :\ B[e_2/x]\end{array}}{\Gamma \vdash (\lambda x : A \,.\, e_1)\, e_2 \equiv_{B[e_2/x]} e_1[e_2/x]}$$

But there is such a thing as being too paranoid. For instance, here we have the β-rule for functions in full paranoia mode. There are 7 premises. We can get rid of some of them. The last two premises ought to follow from general properties of substitution. The first three premises should follow by a suitable meta-theorem from the remaining two.

What we are left with is a possible rule, but not the only one, and not a good one, as it requires re-checking subterms. It is not our purpose here to figure this out, we are just demonstrating what sort of thing one has to be careful about. We can solve problems in another way.

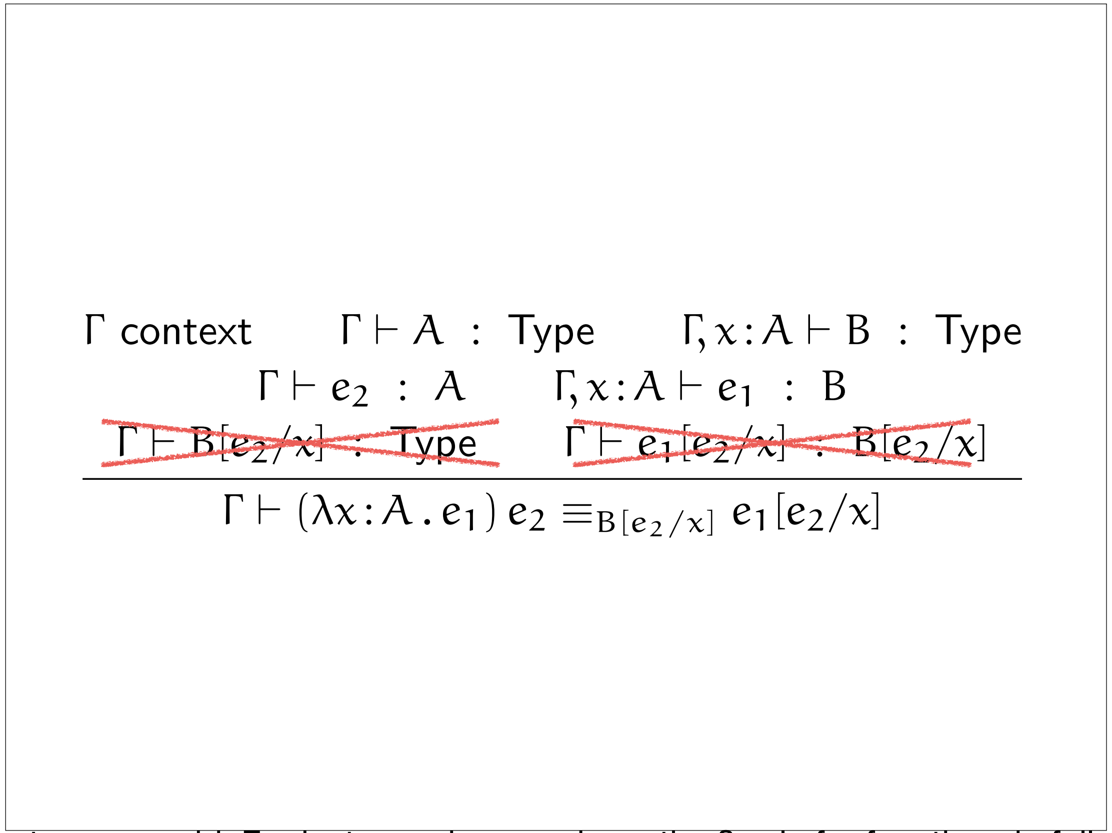


$$\dfrac{\begin{array}{c}\Gamma\ \text{context} \qquad \Gamma \vdash A : \mathsf{Type} \qquad \Gamma, x{:}A \vdash B : \mathsf{Type} \\ \Gamma \vdash e_2 : A \qquad \Gamma, x{:}A \vdash e_1 : B \\ \cancel{\Gamma \vdash B[e_2/x] : \mathsf{Type}} \qquad \cancel{\Gamma \vdash e_1[e_2/x] : B[e_2/x]}\end{array}}{\Gamma \vdash (\lambda x{:}A\,.\,e_1)\, e_2 \equiv_{B[e_2/x]} e_1[e_2/x]}$$

follow from
properties
of substitution

But there is such a thing as being too paranoid. For instance, here we have the β-rule for functions in full paranoia mode. There are 7 premises. We can get rid of some of them. The last two premises ought to follow from general properties of substitution. The first three premises should follow by a suitable meta-theorem from the remaining two.

What we are left with is a possible rule, but not the only one, and not a good one, as it requires re-checking subterms. It is not our purpose here to figure this out, we are just demonstrating what sort of thing one has to be careful about. We can solve problems in another way.



$$\frac{\begin{array}{c} \Gamma\ \text{context} \qquad \Gamma \vdash A \ :\ \mathsf{Type} \qquad \Gamma, x : A \vdash B \ :\ \mathsf{Type} \\ \Gamma \vdash e_2 \ :\ A \qquad \Gamma, x : A \vdash e_1 \ :\ B \\ \xcancel{\Gamma \vdash B[e_2/x] \ :\ \mathsf{Type}} \qquad \xcancel{\Gamma \vdash e_1[e_2/x] \ :\ B[e_2/x]} \end{array}}{\Gamma \vdash (\lambda x : A \,.\, e_1)\, e_2 \equiv_{B[e_2/x]} e_1[e_2/x]}$$

But there is such a thing as being too paranoid. For instance, here we have the β-rule for functions in full paranoia mode. There are 7 premises. We can get rid of some of them. The last two premises ought to follow from general properties of substitution. The first three premises should follow by a suitable meta-theorem from the remaining two.

What we are left with is a possible rule, but not the only one, and not a good one, as it requires re-checking subterms. It is not our purpose here to figure this out, we are just demonstrating what sort of thing one has to be careful about. We can solve problems in another way.

by a meta-theorem

$$\dfrac{\begin{array}{c}\cancel{\Gamma\ \mathsf{context}} \qquad \cancel{\Gamma \vdash A : \mathsf{Type}} \qquad \cancel{\Gamma, x : A \vdash B : \mathsf{Type}} \\ \Gamma \vdash e_2 : A \qquad \Gamma, x : A \vdash e_1 : B \\ \cancel{\Gamma \vdash B[e_2/x] : \mathsf{Type}} \qquad \cancel{\Gamma \vdash e_1[e_2/x] : B[e_2/x]}\end{array}}{\Gamma \vdash (\lambda x : A \,.\, e_1)\, e_2 \equiv_{B[e_2/x]} e_1[e_2/x]}$$

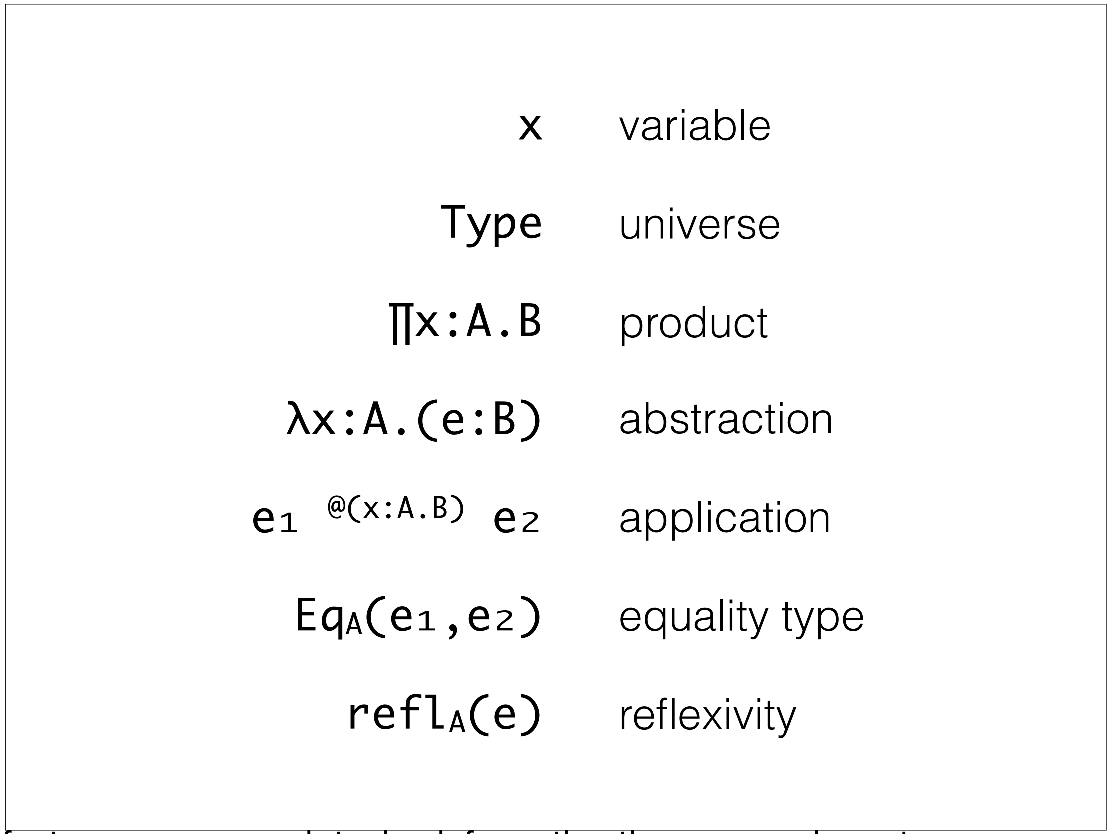
But there is such a thing as being too paranoid. For instance, here we have the β-rule for functions in full paranoia mode. There are 7 premises. We can get rid of some of them. The last two premises ought to follow from general properties of substitution. The first three premises should follow by a suitable meta-theorem from the remaining two.

What we are left with is a possible rule, but not the only one, and not a good one, as it requires re-checking subterms. It is not our purpose here to figure this out, we are just demonstrating what sort of thing one has to be careful about. We can solve problems in another way.

$$\frac{\begin{array}{c}\xcancel{\Gamma\ \mathsf{context}} \qquad \xcancel{\Gamma \vdash A : \mathsf{Type}} \qquad \xcancel{\Gamma, x{:}A \vdash B : \mathsf{Type}} \\ \Gamma \vdash e_2 : A \qquad \Gamma, x{:}A \vdash e_1 : B \\ \xcancel{\Gamma \vdash B[e_2/x] : \mathsf{Type}} \qquad \xcancel{\Gamma \vdash e_1[e_2/x] : B[e_2/x]}\end{array}}{\Gamma \vdash (\lambda x{:}A \,.\, e_1)\, e_2 \equiv_{B[e_2/x]} e_1[e_2/x]}$$

But there is such a thing as being too paranoid. For instance, here we have the β-rule for functions in full paranoia mode. There are 7 premises. We can get rid of some of them. The last two premises ought to follow from general properties of substitution. The first three premises should follow by a suitable meta-theorem from the remaining two.

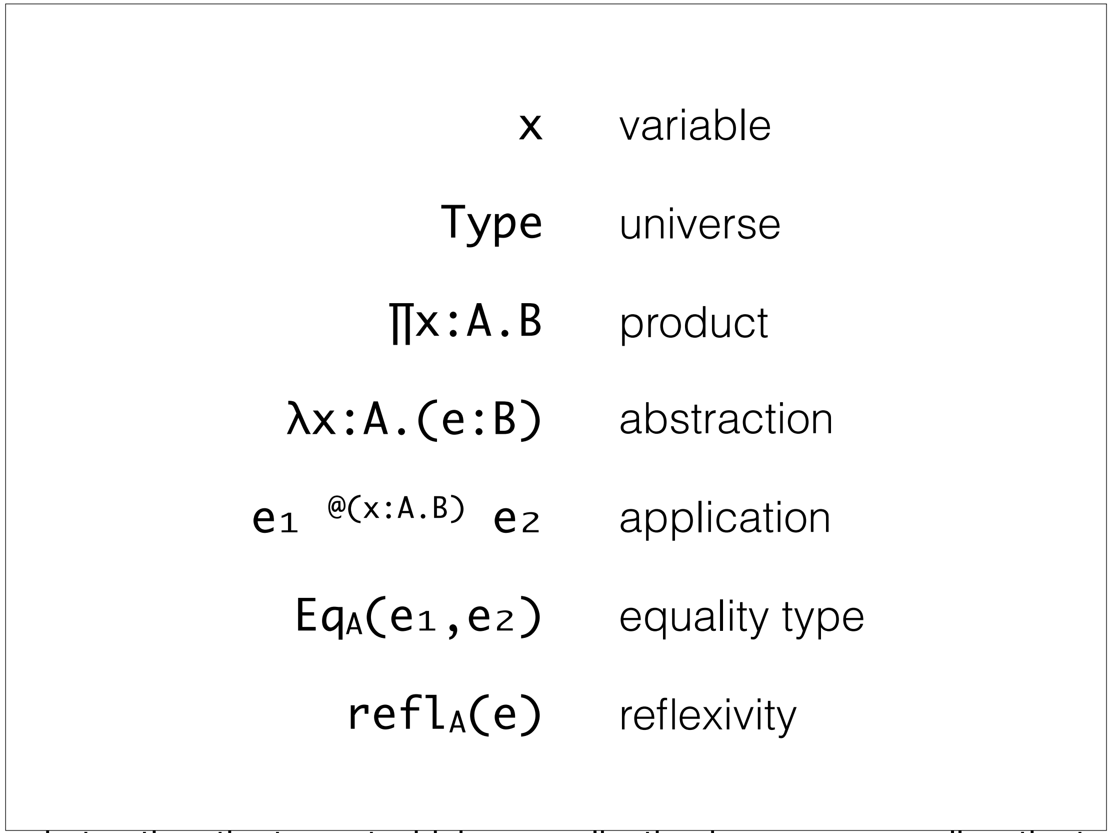
What we are left with is a possible rule, but not the only one, and not a good one, as it requires re-checking subterms. It is not our purpose here to figure this out, we are just demonstrating what sort of thing one has to be careful about. We can solve problems in another way.



| | |
|---|---|
| $x$ | variable |
| $\mathtt{Type}$ | universe |
| $\prod x:A.B$ | product |
| $\lambda x:A.(e:B)$ | abstraction |
| $e_1 \ {}^{@(x:A.B)}\ e_2$ | application |
| $Eq_A(e_1,e_2)$ | equality type |
| $refl_A(e)$ | reflexivity |

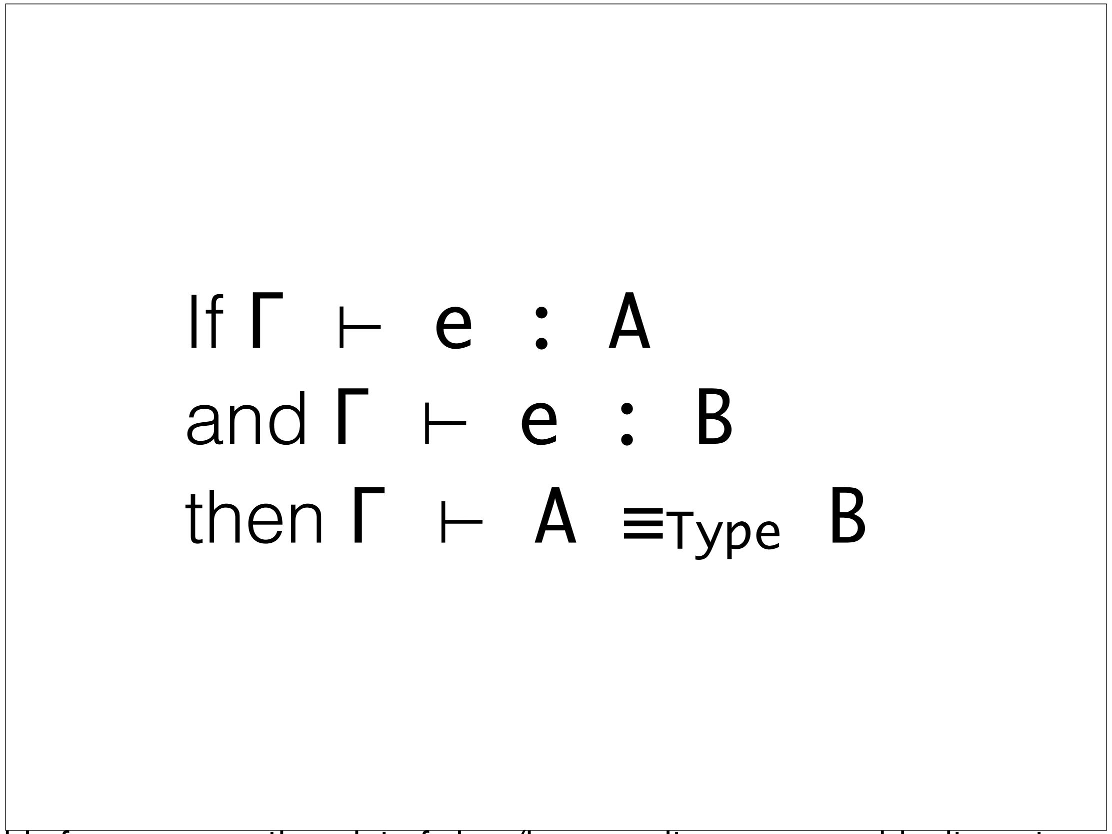
If we make sure that the terms of a type carry enough typing information then we can hope to prove a couple of meta-theorems that will make life easier.

So we can put extra information about types in with *typing annotations.*



| | |
|---|---|
| $\mathtt{x}$ | variable |
| $\mathtt{Type}$ | universe |
| $\prod \mathtt{x:A.B}$ | product |
| $\lambda \mathtt{x:A.(e:B)}$ | abstraction |
| $\mathtt{e_1} \ ^{\mathtt{@(x:A.B)}} \ \mathtt{e_2}$ | application |
| $\mathtt{Eq_A(e_1,e_2)}$ | equality type |
| $\mathtt{refl_A(e)}$ | reflexivity |

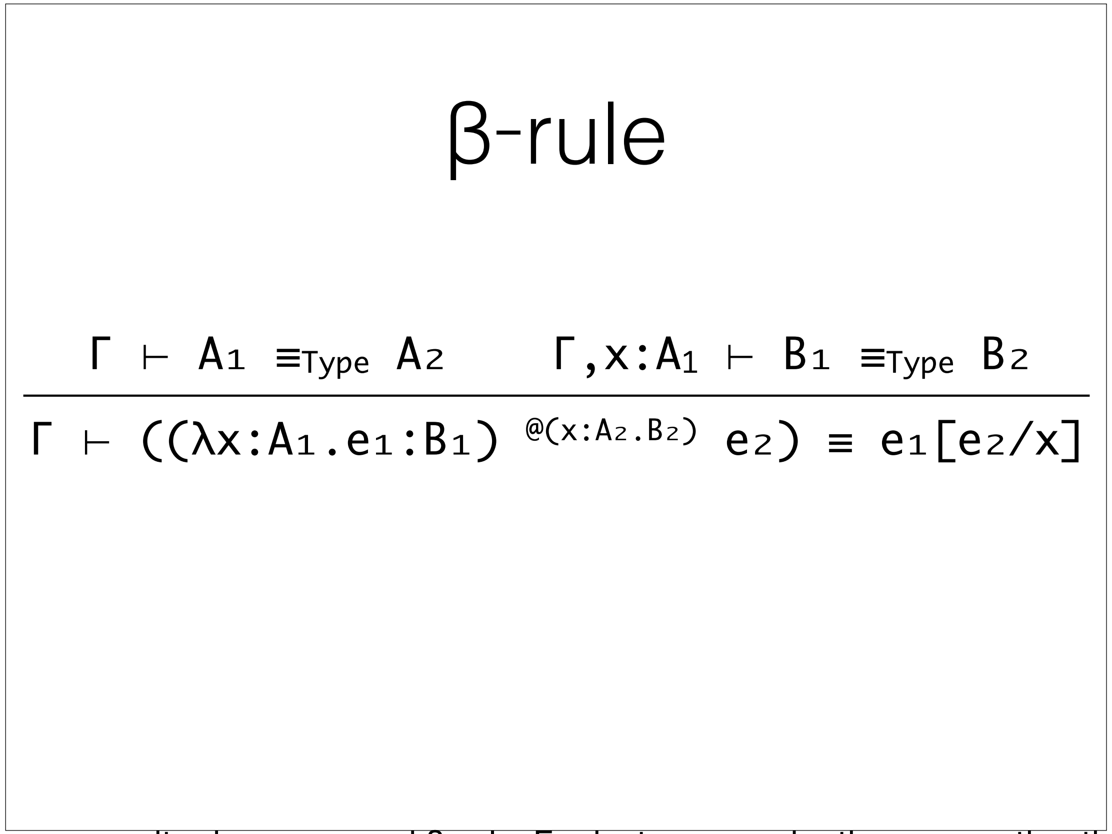
We indicate the output type of an abstraction, the type at which an application happens, as well as the type at which equality is considered. This way the meaning of a well-formed term is clear from the term itself.



If $\Gamma \vdash \mathtt{e} : \mathtt{A}$
and $\Gamma \vdash \mathtt{e} : \mathtt{B}$
then $\Gamma \vdash \mathtt{A} \equiv_{\mathtt{Type}} \mathtt{B}$

One meta-theorem that is desirable from a semantic point of view (because it removes ambiguity as to possible meanings of terms) is *uniqueness of typing*.

One can think of other meta-theorems that are desirable, but we shall not go into this here. It is a perfect job for a PhD student, as it involves proving lots of boring but finicky structural inductions (or use of logical relations if the student is unlucky).



## β-rule

$$\frac{\Gamma \vdash A_1 \equiv_{\mathrm{Type}} A_2 \qquad \Gamma, x{:}A_1 \vdash B_1 \equiv_{\mathrm{Type}} B_2}{\Gamma \vdash ((\lambda x{:}A_1.e_1 : B_1)\ {}^{@(x:A_2.B_2)}\ e_2) \equiv e_1[e_2/x]}$$

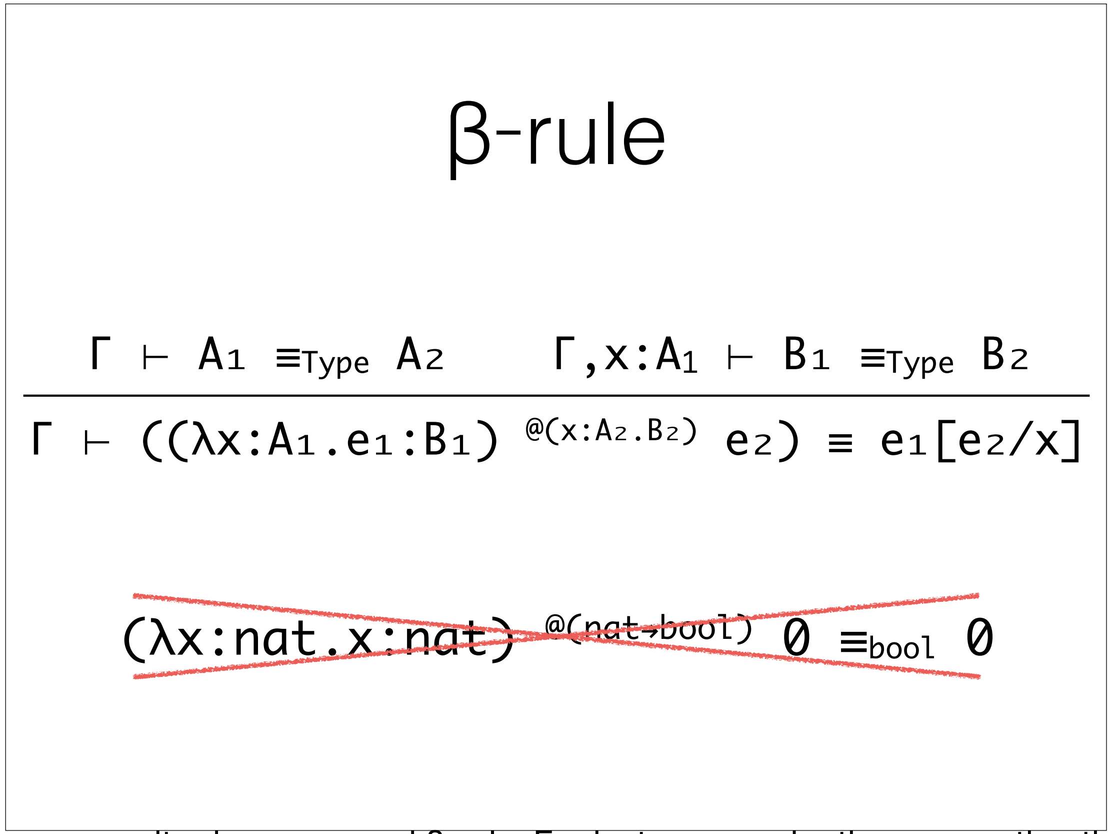
With explicit typing annotations we can write down a sound β-rule. For instance, under the assumption that $\mathrm{nat} \to \mathrm{bool} \equiv \mathrm{nat} \to \mathrm{nat}$ the abstraction on the left-hand side is well-formed, but the β-rule does not apply because types mismatch.
Let us turn attention to how we could give some *meaning* to types.

## β-rule

$$\frac{\Gamma \vdash A_1 \equiv_{Type} A_2 \qquad \Gamma, x{:}A_1 \vdash B_1 \equiv_{Type} B_2}{\Gamma \vdash ((\lambda x{:}A_1.e_1{:}B_1)\ ^{@(x:A_2.B_2)}\ e_2) \equiv e_1[e_2/x]}$$

$$(\lambda x{:}nat.x{:}nat)\ ^{@(nat\to bool)}\ 0 \equiv_{bool} 0$$

With explicit typing annotations we can write down a sound β-rule. For instance, under the assumption that nat→bool ≡ nat → nat the abstraction on the left-hand side is well-formed, but the β-rule does not apply because types mismatch.
Let us turn attention to how we could give some *meaning* to types.

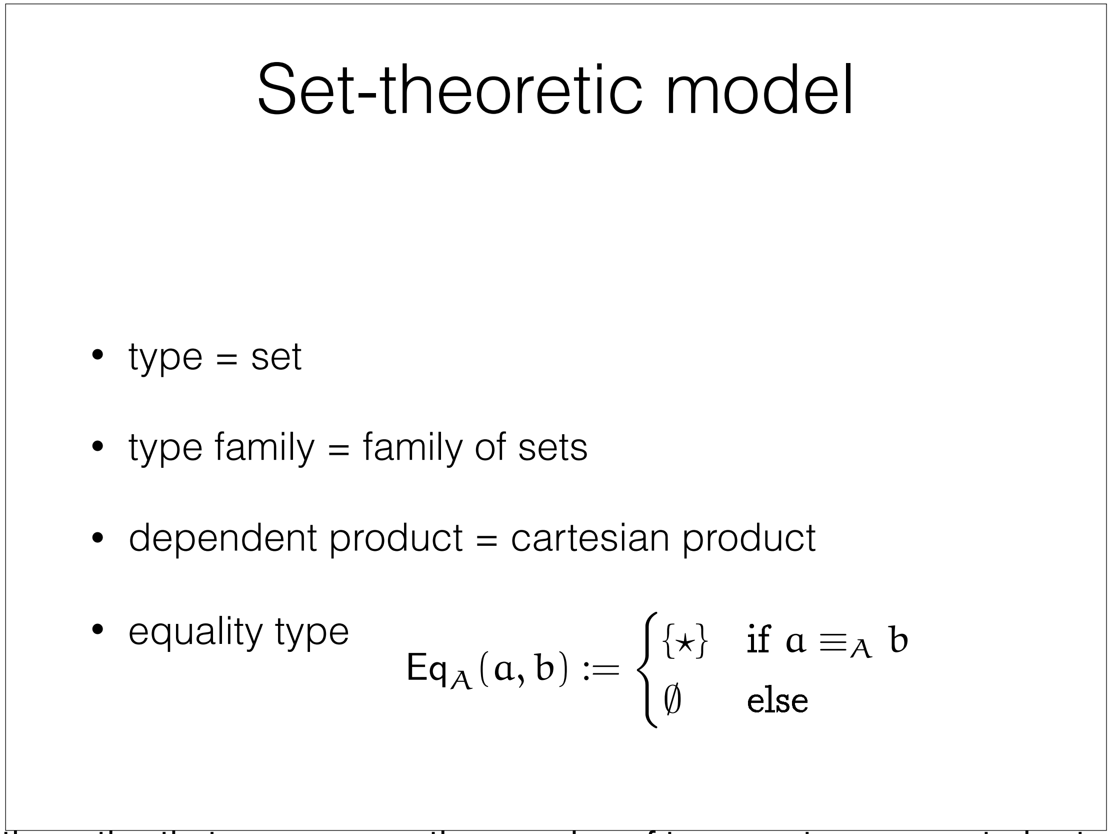


## β-rule

$$\frac{\Gamma \vdash A_1 \equiv_{\mathtt{Type}} A_2 \qquad \Gamma, x{:}A_1 \vdash B_1 \equiv_{\mathtt{Type}} B_2}{\Gamma \vdash ((\lambda x{:}A_1.e_1{:}B_1)\ ^{@(x:A_2.B_2)}\ e_2) \equiv e_1[e_2/x]}$$

$$\xcancel{(\lambda x{:}\mathtt{nat}.x{:}\mathtt{nat})\ ^{@(\mathtt{nat}\to\mathtt{bool})}\ 0 \equiv_{\mathtt{bool}} 0}$$

With explicit typing annotations we can write down a sound β-rule. For instance, under the assumption that nat→bool ≡ nat → nat the abstraction on the left-hand side is well-formed, but the β-rule does not apply because types mismatch.
Let us turn attention to how we could give some *meaning* to types.



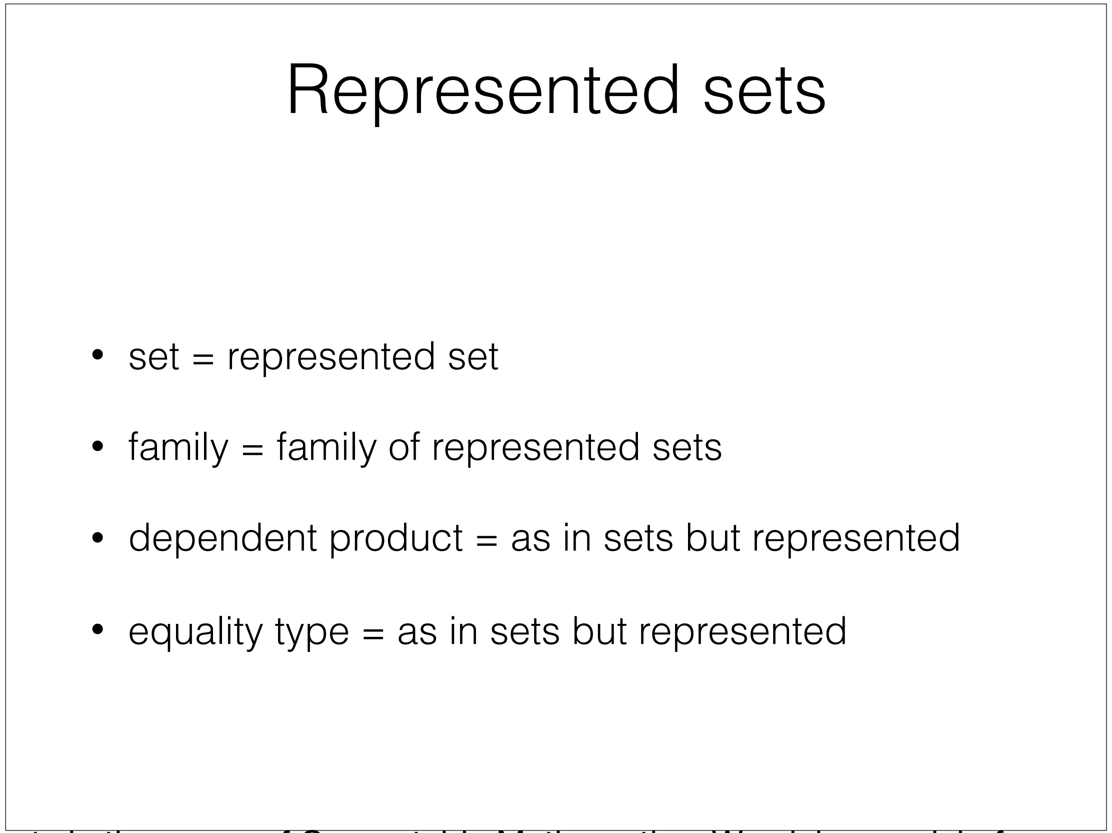
## Set-theoretic model

- type = set
- type family = family of sets
- dependent product = cartesian product
- equality type

$$\mathsf{Eq}_A(a,b) := \begin{cases} \{\star\} & \text{if } a \equiv_A b \\ \emptyset & \text{else} \end{cases}$$

There are many structures in mathematics that can serve as the meaning of types: sets, represented sets, preshaves, sheaves, etc. Such *models* of type theory help develop our understanding of what types are about. (If you have a smart student you can perform an experiment: expose the student *only* to type theory, and watch them develop innate intuition about types. The result of one such experiment will be presented later this week.)

Perhaps the most direct model is the set-theoretic interpretation of types. (Details given on the blackboard.)

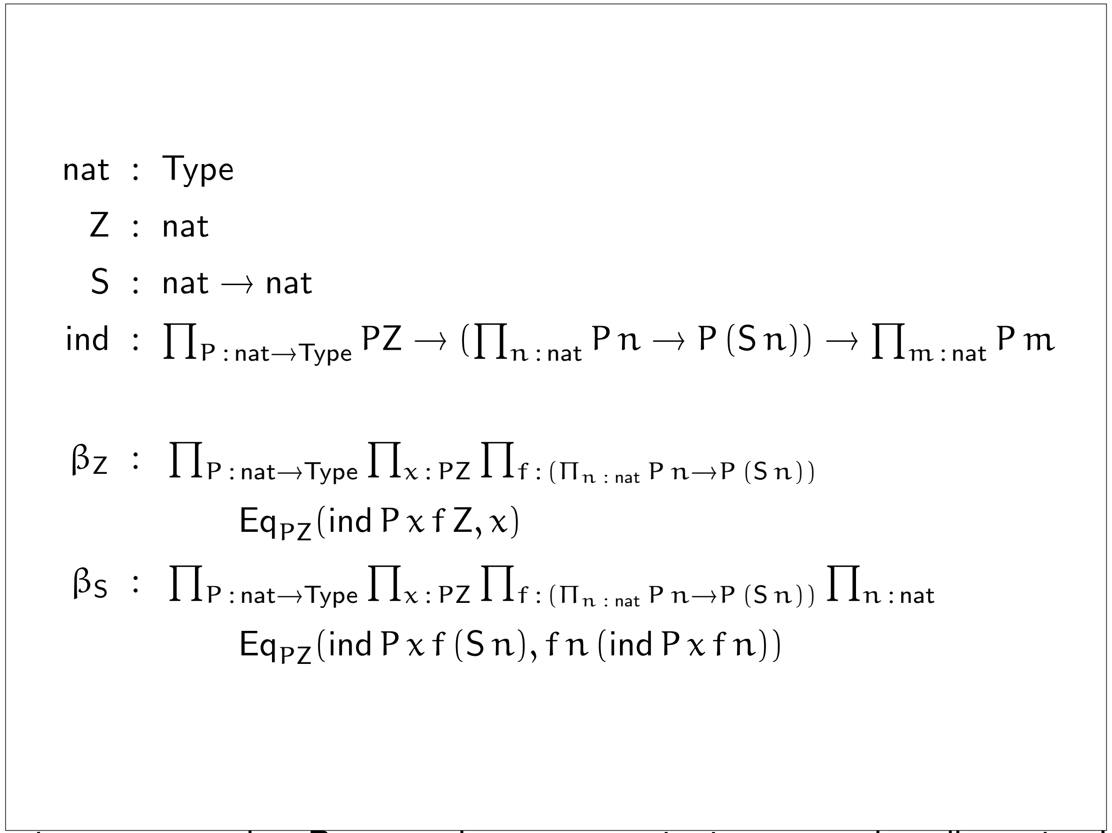


## Represented sets

- set = represented set
- family = family of represented sets
- dependent product = as in sets but represented
- equality type = as in sets but represented

The other model is represented sets in the sense of Computable Mathematics. We pick a model of computation (Turing machines, Haskell programs), and interpret types as sets whose elements are represented in the computational model. (Details given on the blackboard.)

Such models have a lot of extra structure: represented sets have not only equality types and dependent products, but also dependent sums, inductive and coinductive types, quotients, natural numbers, real numbers, and so on. Is our type theory expressive enough to talk about such gadgets, or do we need to extend it?

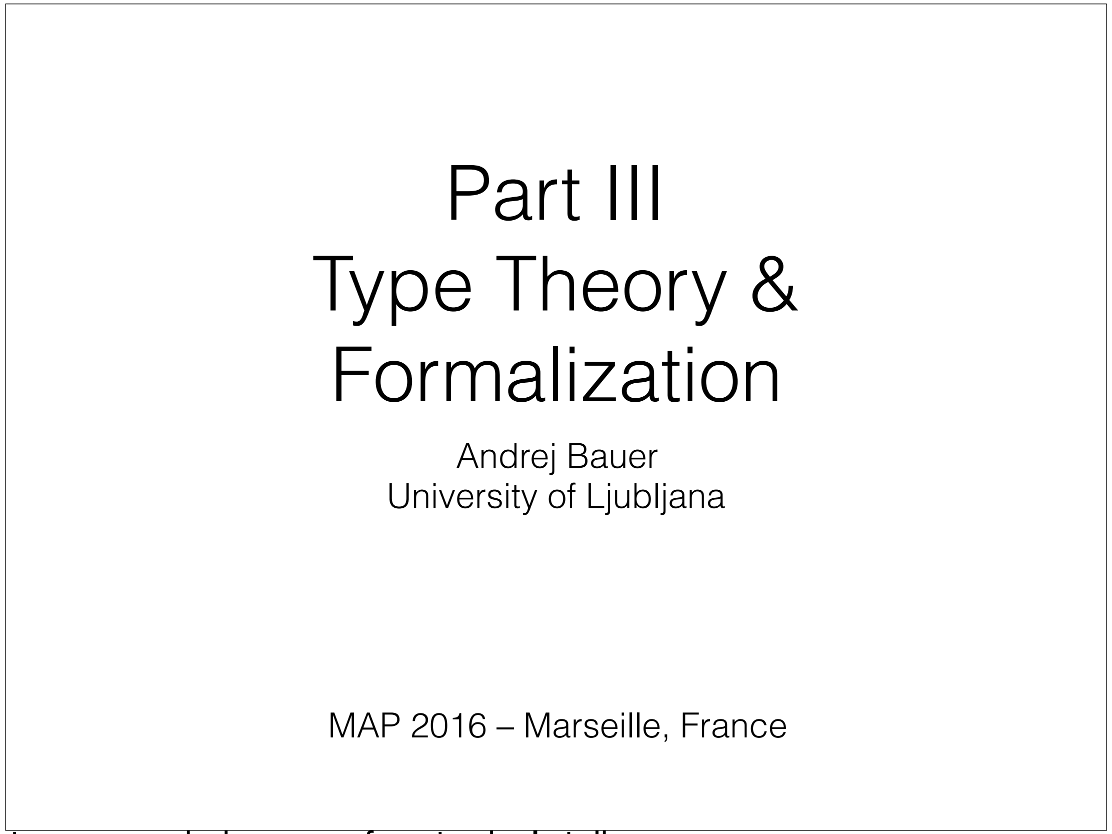


$$\begin{aligned}
\mathsf{nat} &: \mathsf{Type} \\
\mathsf{Z} &: \mathsf{nat} \\
\mathsf{S} &: \mathsf{nat} \to \mathsf{nat} \\
\mathsf{ind} &: \textstyle\prod_{P : \mathsf{nat} \to \mathsf{Type}} P\mathsf{Z} \to (\prod_{n : \mathsf{nat}} P\,n \to P\,(\mathsf{S}\,n)) \to \prod_{m : \mathsf{nat}} P\,m \\
\\
\beta_{\mathsf{Z}} &: \textstyle\prod_{P : \mathsf{nat} \to \mathsf{Type}} \prod_{x : P\mathsf{Z}} \prod_{f : (\Pi_{n : \mathsf{nat}} P\,n \to P\,(\mathsf{S}\,n))} \\
&\qquad \mathsf{Eq}_{P\mathsf{Z}}(\mathsf{ind}\,P\,x\,f\,\mathsf{Z}, x) \\
\beta_{\mathsf{S}} &: \textstyle\prod_{P : \mathsf{nat} \to \mathsf{Type}} \prod_{x : P\mathsf{Z}} \prod_{f : (\Pi_{n : \mathsf{nat}} P\,n \to P\,(\mathsf{S}\,n))} \prod_{n : \mathsf{nat}} \\
&\qquad \mathsf{Eq}_{P\mathsf{Z}}(\mathsf{ind}\,P\,x\,f\,(\mathsf{S}\,n), f\,n\,(\mathsf{ind}\,P\,x\,f\,n))
\end{aligned}$$

It turns out that the theory is in fact very expressive. By assuming *new constants* we can describe not only types, their introduction and elimination forms, but with through reflection also the equalities that these should satisfy. For instance, here is a familiar axiomatization of natural numbers, written in an unreadable form. Many other constructions can be described in a similar way.

This is all very good, but the full syntax of type theory with typing annotations (which are missing in the axiomatization of natural numbers) is completely impractical for humans. We face a design question: make type theory usable for humans. We shall say a bit about this in the last lecture.

End of Part II

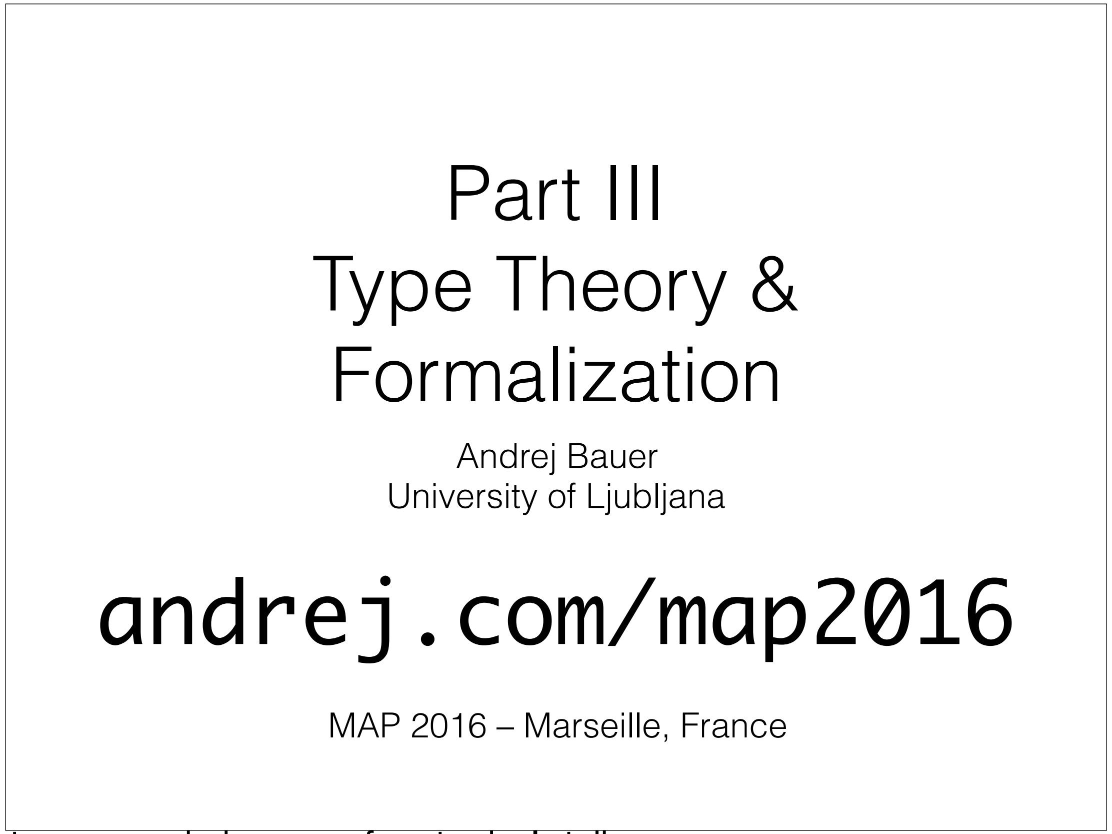


# Part III Type Theory & Formalization

Andrej Bauer
University of Ljubljana

MAP 2016 – Marseille, France

Let us first deal with an issue that came up during one of yesterday's talks.



# Part III
# Type Theory & Formalization

Andrej Bauer
University of Ljubljana

**andrej.com/map2016**

MAP 2016 – Marseille, France

Let us first deal with an issue that came up during one of yesterday's talks.

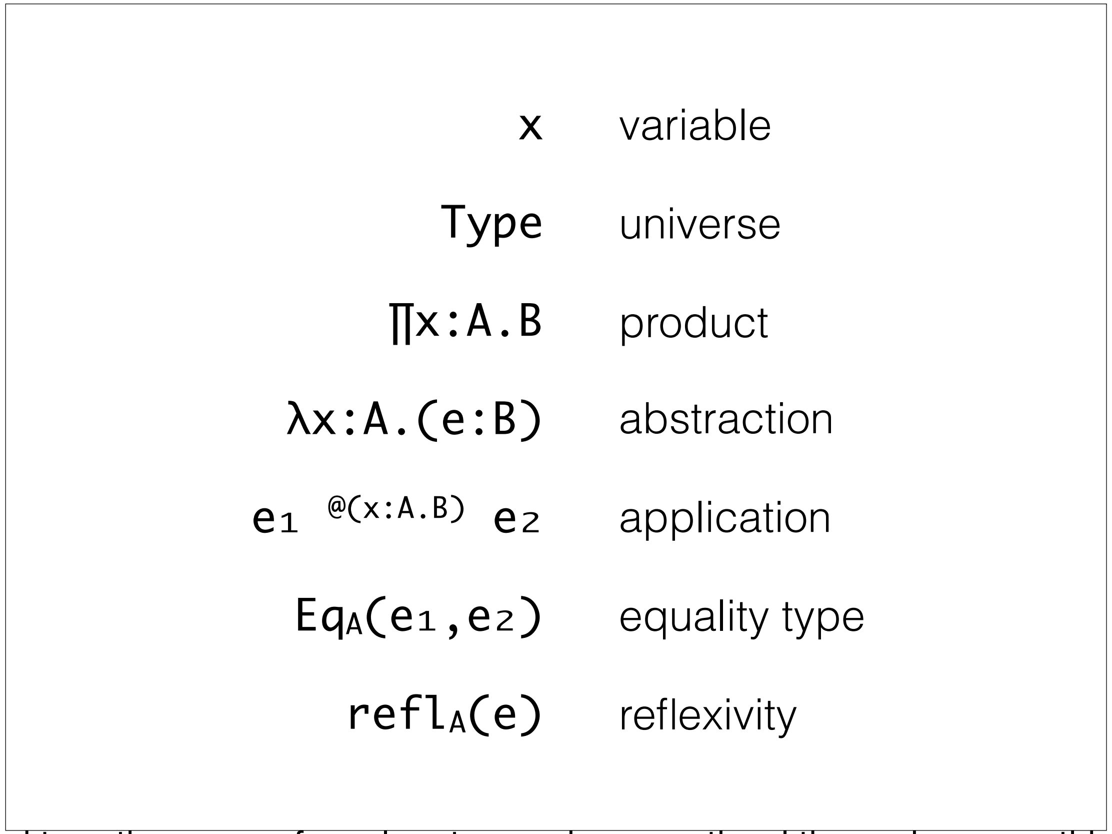
Chuck Norris can divide by zero.

Chuck Norris can slam a revolving door.

Chuck Norris knows Victoria's Secret.

It’s true. (Source: http://www.chucknorrisfacts.com)

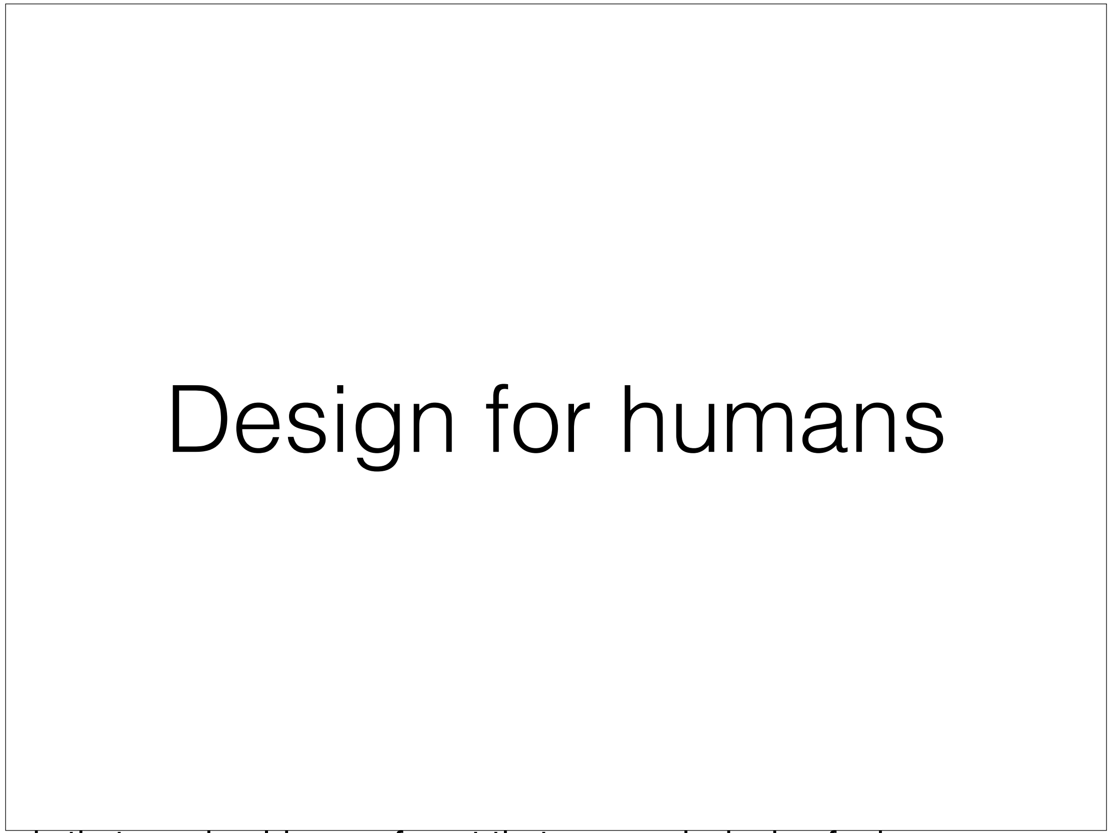
Here are some Chuck Norris jokes from http://www.chucknorrisfacts.com.



| | |
|---|---|
| `x` | variable |
| `Type` | universe |
| `∏x:A.B` | product |
| `λx:A.(e:B)` | abstraction |
| $e_1 \ ^{@(x:A.B)} \ e_2$ | application |
| $\mathrm{Eq}_A(e_1,e_2)$ | equality type |
| $\mathrm{refl}_A(e)$ | reflexivity |

In the last two lectures we studied type theory as a formal system and an equational theory, i.e., something that is within the realm of abstract mathematics. We saw that the details of type theory can be quite finicky and not something that we can trust ourselves to do correctly by hand. It would be helpful to get some support from computers. Today's question is: how do we *implement* type theory?

There is a long and venerable history of building powerful theorem provers and proof assistants, which have been put to impressive use by participants of this meeting. How are they designed? The answer to this question would take a whole course. We shall look just at some guiding principles. Then we will apply them to our minimalist type theory.



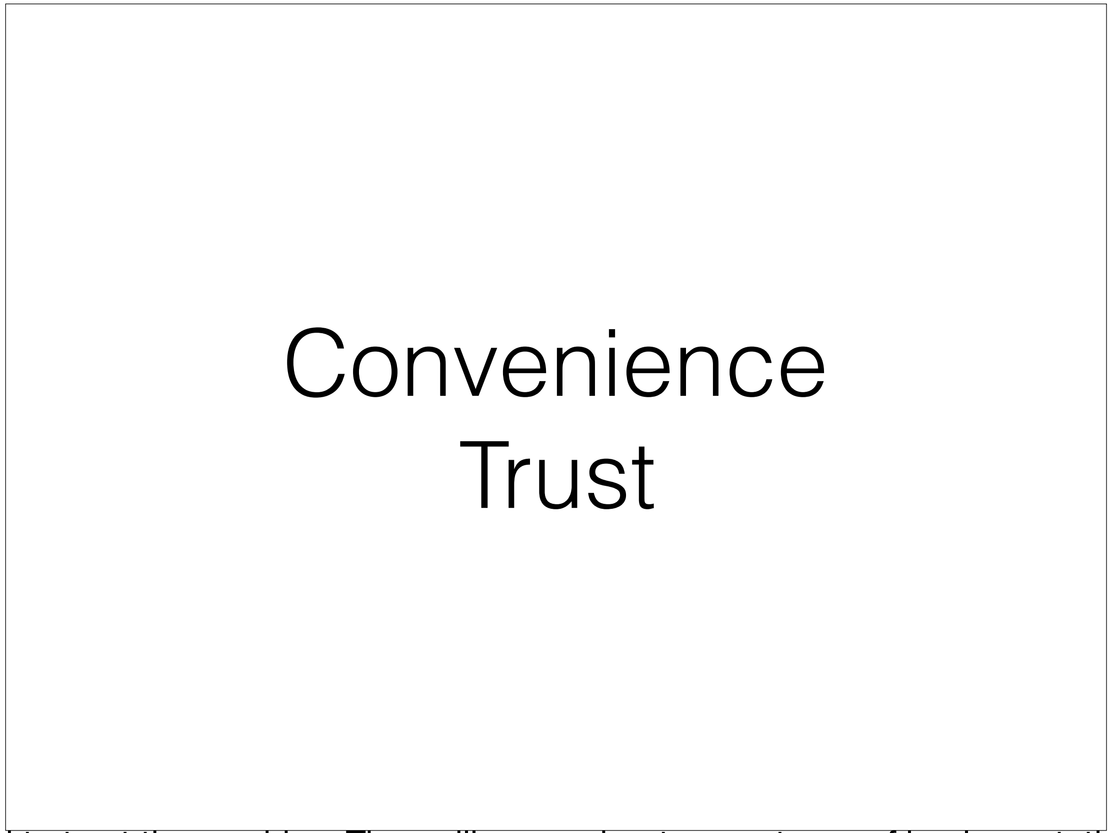
## Design for humans

I emphasized in the first talk already that we should never forget that we are designing for humans.

Human are difficult to please.

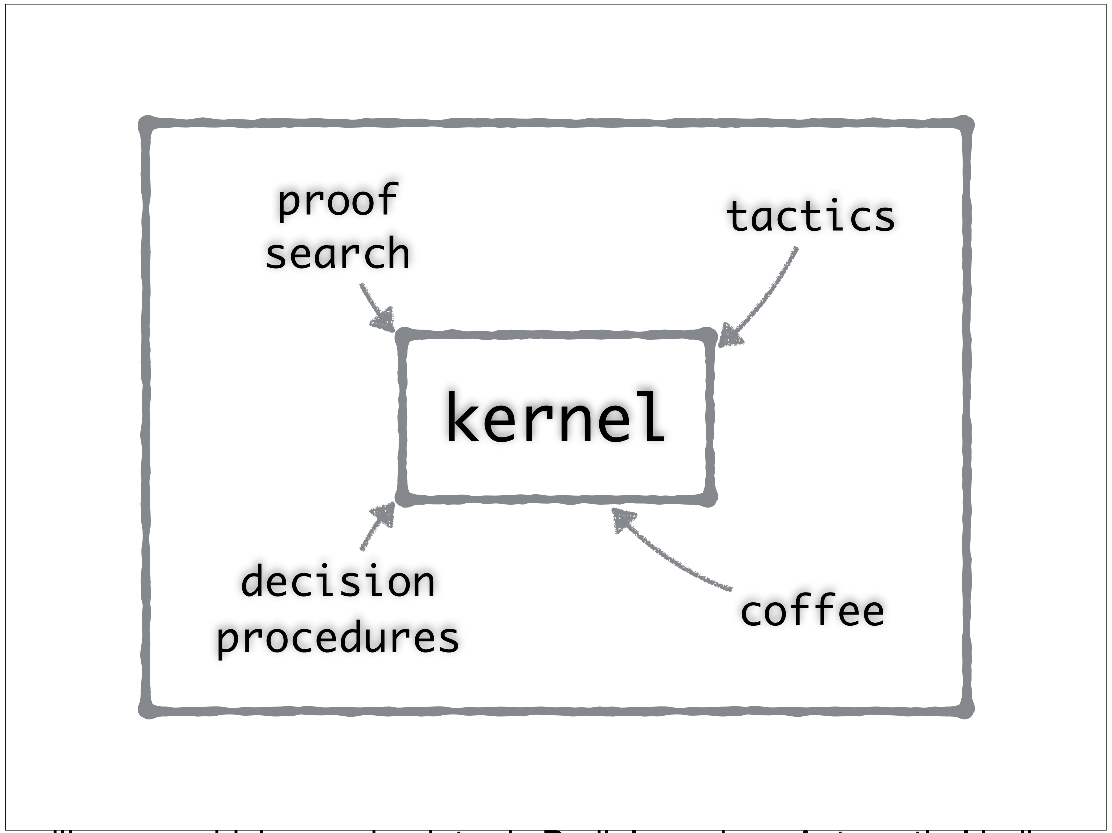
## Convenience

They want convenience: the machine should do everything, the machine should guess what the human means, the machine should be smart. They want a *complex* system.

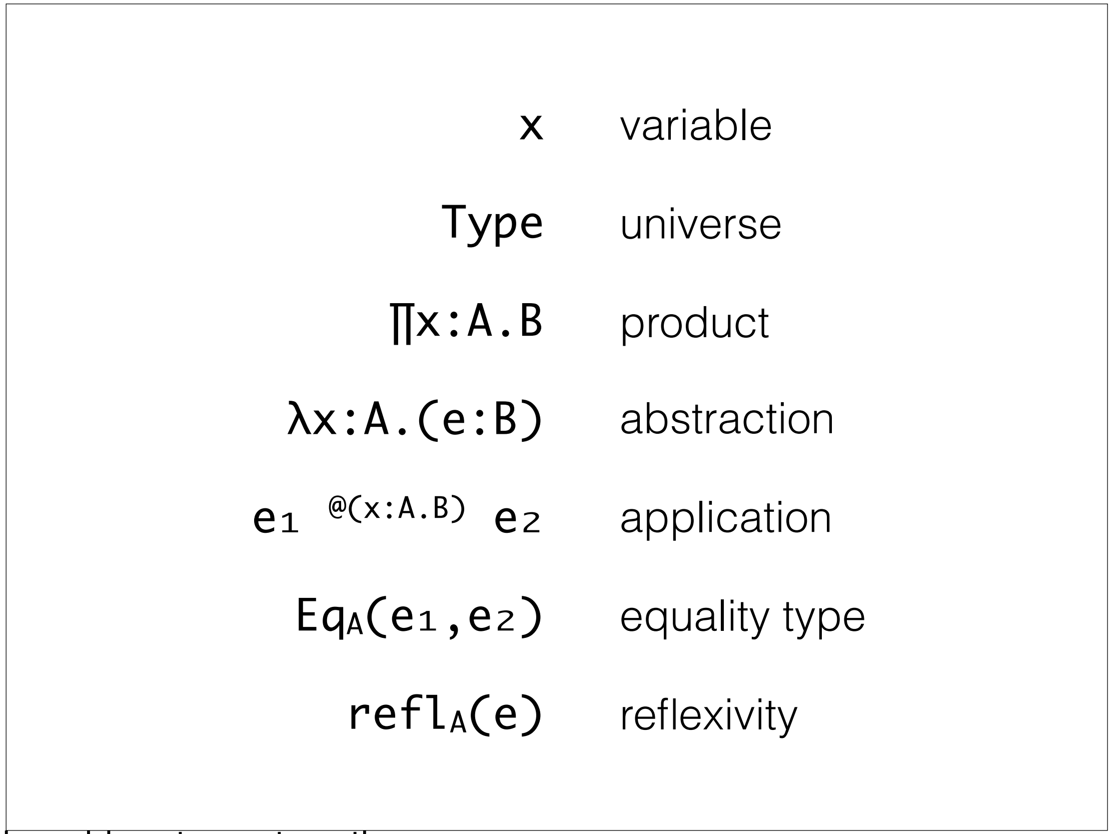


## Convenience
## Trust

On the other hand, humans need to trust the machine. They will worry about correctness of implementation. They will more readily trust a *simple* system.



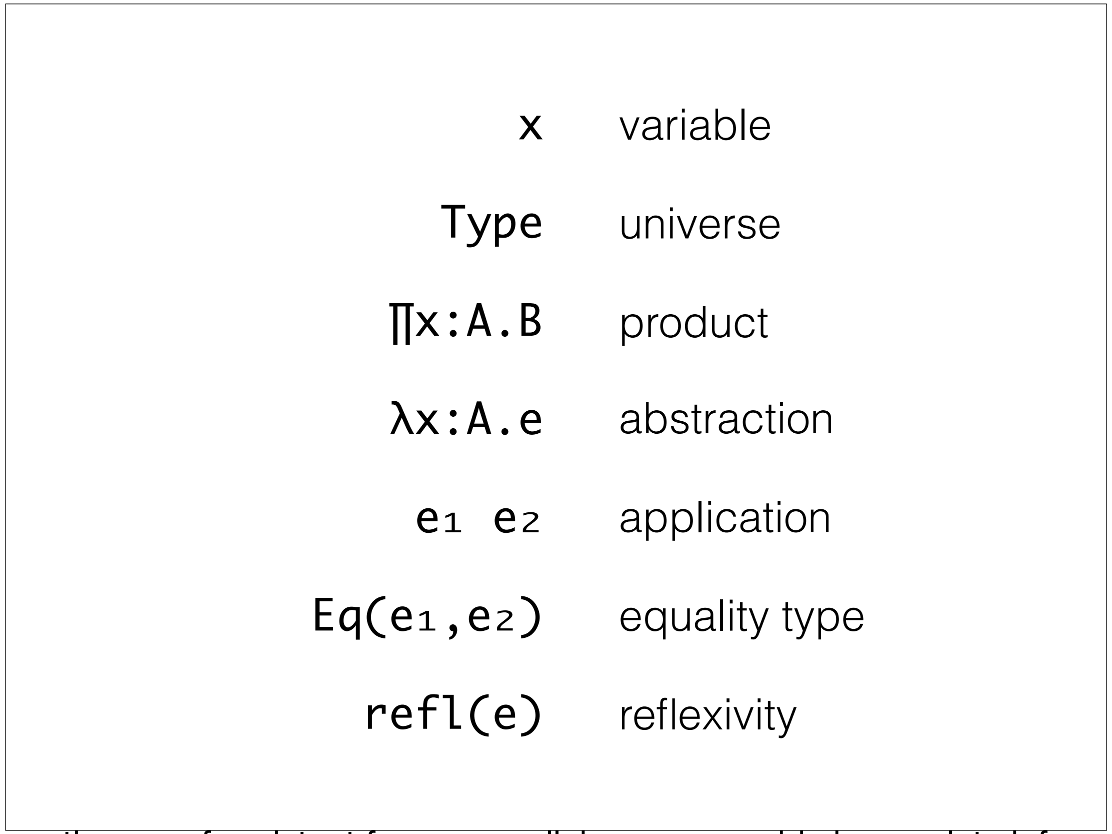
There is a standard solution to the dilemma, which goes back to de Bruijn's work on Automath, I believe. A proof system (by which I mean a proof assistant, a theorem prover, or any system that helps verify or discover proofs) should be designed in such a way that all proofs pass through a verifier, called the *kernel.* Then we can trust the whole system as long as we trust the kernel, which therefore ought to be as simple as possible. For instance John Harrison, the author of HOL/light, has been seen walking around confrences with the HOL/light kernel printed on his T-shirt.



| | |
|---|---|
| `x` | variable |
| `Type` | universe |
| `∏x:A.B` | product |
| `λx:A.(e:B)` | abstraction |
| $e_1 \;{}^{@(x:A.B)}\; e_2$ | application |
| $\mathrm{Eq}_A(e_1,e_2)$ | equality type |
| $\mathrm{refl}_A(e)$ | reflexivity |

Let us see how we could apply these ideas to our type theory.

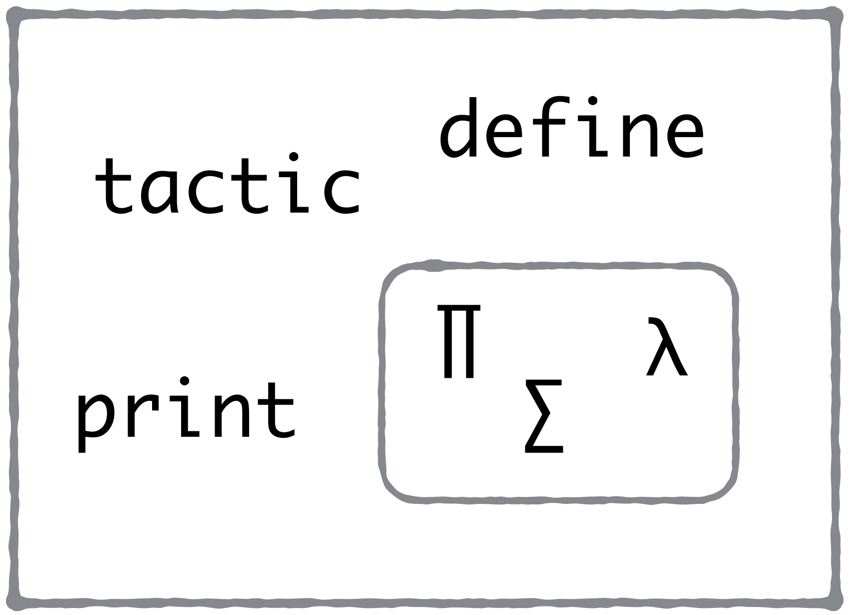
Firstly, we expect that the user will be unwilling to type all the annotations.



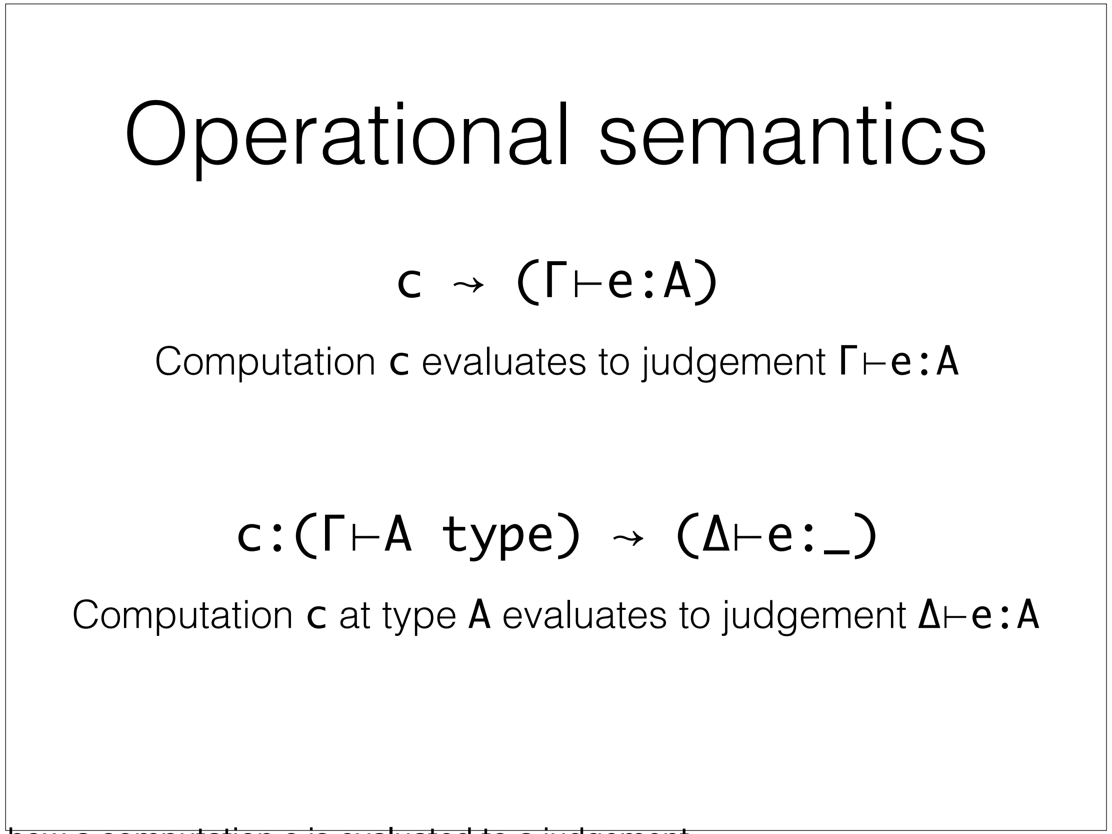
| | |
|---|---|
| $\mathtt{x}$ | variable |
| $\mathtt{Type}$ | universe |
| $\prod \mathtt{x{:}A.B}$ | product |
| $\lambda \mathtt{x{:}A.e}$ | abstraction |
| $\mathtt{e}_1\ \mathtt{e}_2$ | application |
| $\mathtt{Eq}(\mathtt{e}_1,\mathtt{e}_2)$ | equality type |
| $\mathtt{refl}(\mathtt{e})$ | reflexivity |

So we face the problem that every other proof assistant faces as well: humans provide incomplete information *already at the level of types.* Not only is it the case that the human is not providing a detailed construction, they are not even telling you *what* they want to construct. If you observe how mathematicians communicate, you will notice that they indeed *reconstruct missing information all the time*, and the proof assistant should do so as well.

The reconstruction of missing information has to be *predictable*, but experience shows that it is also *complex*. (Think of mechanisms such as implicit arguments, types classes and canonical structures.) We shall make the kernel *predictable* but *flexible* so that users can then implement complex guessing techniques.



We shall think of the kernel as an *evaluator* and the input language as a *programming language*. Many systems do this: HOL has OCaml as the ambient language, Coq has its own tactic language, etc. Under this view the type theory is *contained* in the surrounding programming language.



## Operational semantics

`c → (Γ⊢e:A)`

Computation `c` evaluates to judgement `Γ⊢e:A`

`c:(Γ⊢A type) → (Δ⊢e:_)`

Computation `c` at type `A` evaluates to judgement `Δ⊢e:A`

The operational semantics explains how a computation c is evaluated to a judgement.

There are two modes, they correspond to the inferring and checking mode known in bidirectional type checking.

## Operational semantics

“inferring” mode

$$\texttt{c} \to (\Gamma\vdash \texttt{e:A})$$

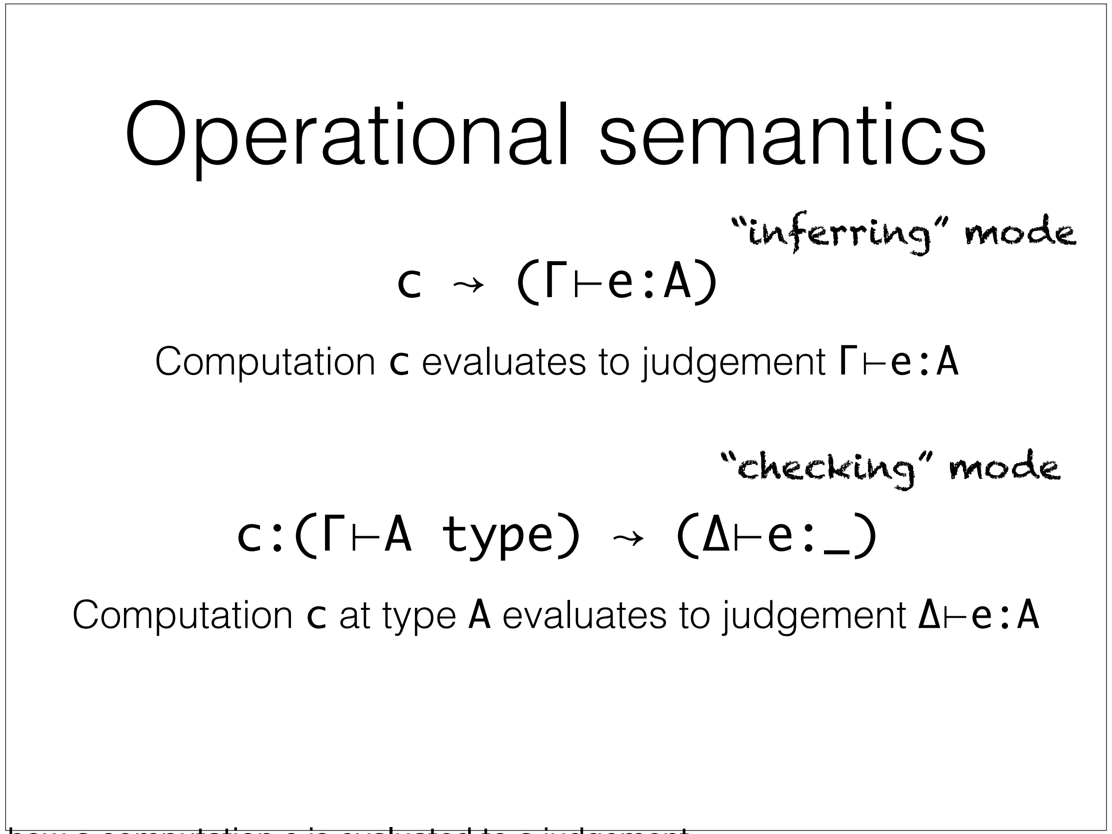
Computation `c` evaluates to judgement $\Gamma\vdash$`e:A`

$$\texttt{c:}(\Gamma\vdash \texttt{A type}) \to (\Delta\vdash \texttt{e:\_})$$

Computation `c` at type `A` evaluates to judgement $\Delta\vdash$`e:A`

The operational semantics explains how a computation c is evaluated to a judgement.

There are two modes, they correspond to the inferring and checking mode known in bidirectional type checking.



## Operational semantics

“inferring” mode

`c → (Γ⊢e:A)`

Computation `c` evaluates to judgement `Γ⊢e:A`

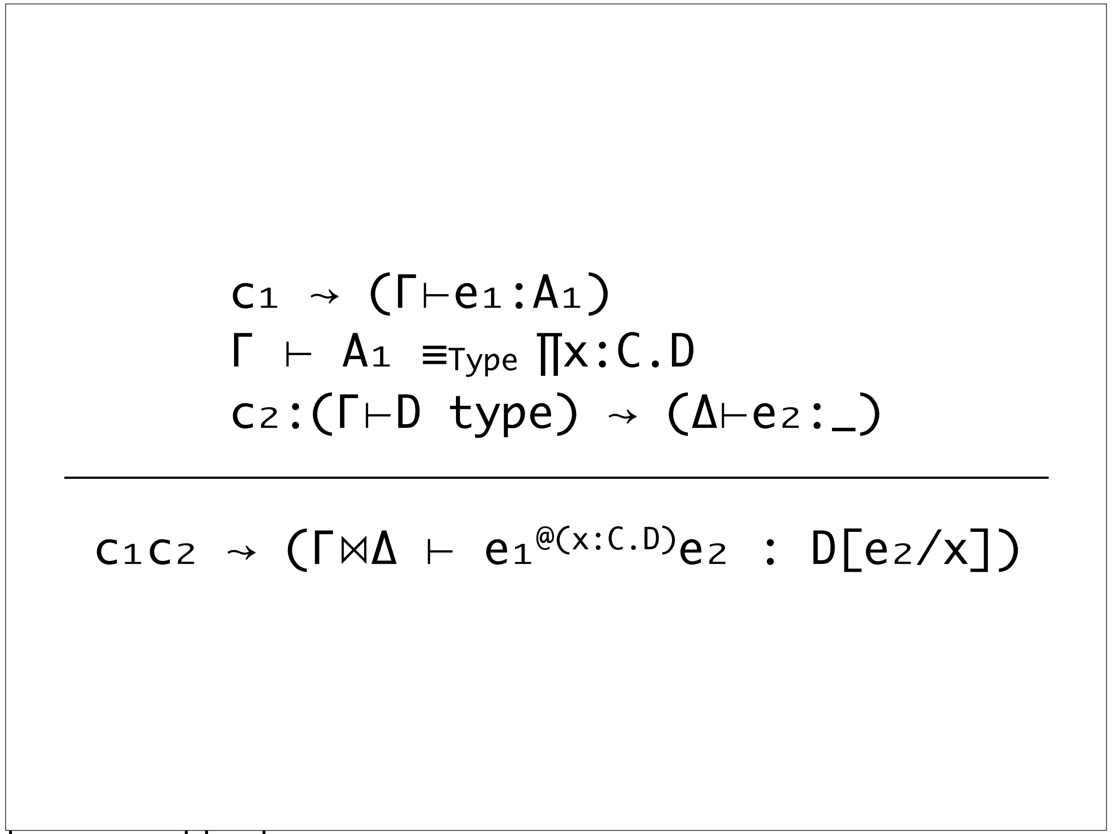
“checking” mode

`c:(Γ⊢A type) → (Δ⊢e:_)`

Computation `c` at type `A` evaluates to judgement `Δ⊢e:A`

The operational semantics explains how a computation c is evaluated to a judgement.

There are two modes, they correspond to the inferring and checking mode known in bidirectional type checking.

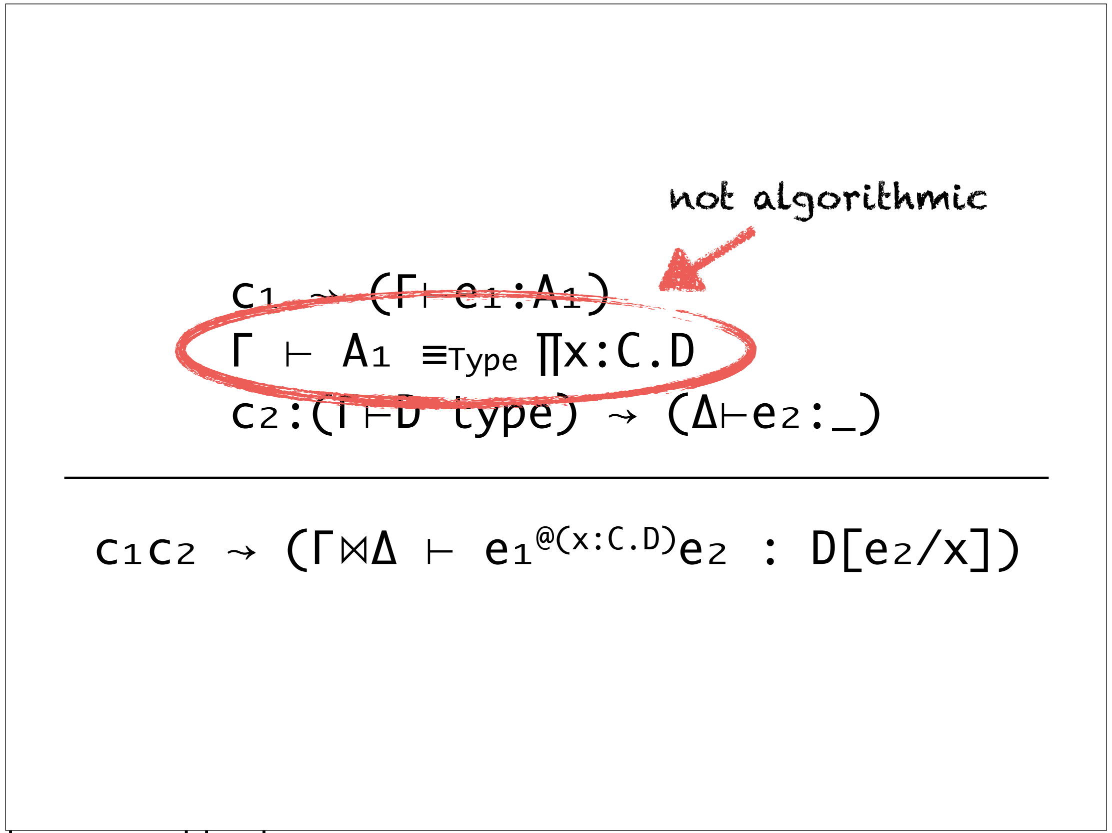


$$\frac{\begin{array}{l} c_1 \to (\Gamma \vdash e_1 : A_1) \\ \Gamma \vdash A_1 \equiv_{\mathrm{Type}} \prod x{:}C.D \\ c_2 : (\Gamma \vdash D \text{ type}) \to (\Delta \vdash e_2 : \_) \end{array}}{c_1 c_2 \to (\Gamma \bowtie \Delta \vdash e_1^{@(x:C.D)} e_2 : D[e_2/x])}$$

An example on how the two modes are combined.

There are some problems. First of all, how do we make sure that $A_1$ is a product? In nice type theories the answer would be: normalize $A_1$ and see if you get a product. But we cannot do that here because our type theory is *not* normalizing. In fact, **recognizing the shape of a type is a fundamental activity** that is performed by mathematicians. It is not captured by traditional rules of type theory.

And we need to combine the different contexts, which may not actually be compatible with each other.



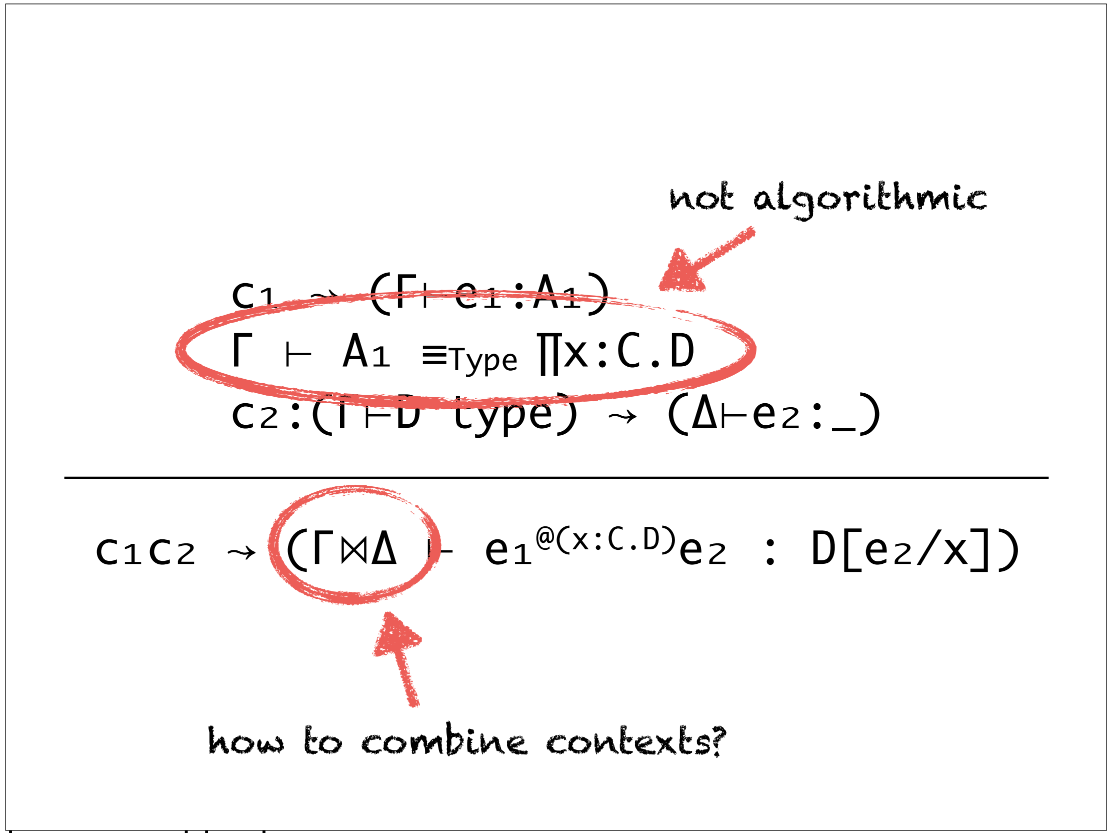
not algorithmic

$$\frac{\begin{array}{l} c_1 \to (\Gamma \vdash e_1 : A_1) \\ \Gamma \vdash A_1 \equiv_{\mathsf{Type}} \prod x{:}C.D \\ c_2 : (\Gamma \vdash D\ \mathsf{type}) \to (\Delta \vdash e_2 : \_) \end{array}}{c_1 c_2 \to (\Gamma \bowtie \Delta \vdash e_1{}^{@(x:C.D)} e_2 : D[e_2/x])}$$

An example on how the two modes are combined.

There are some problems. First of all, how do we make sure that $A_1$ is a product? In nice type theories the answer would be: normalize $A_1$ and see if you get a product. But we cannot do that here because our type theory is *not* normalizing. In fact, **recognizing the shape of a type is a fundamental activity** that is performed by mathematicians. It is not captured by traditional rules of type theory.

And we need to combine the different contexts, which may not actually be compatible with each other.


not algorithmic

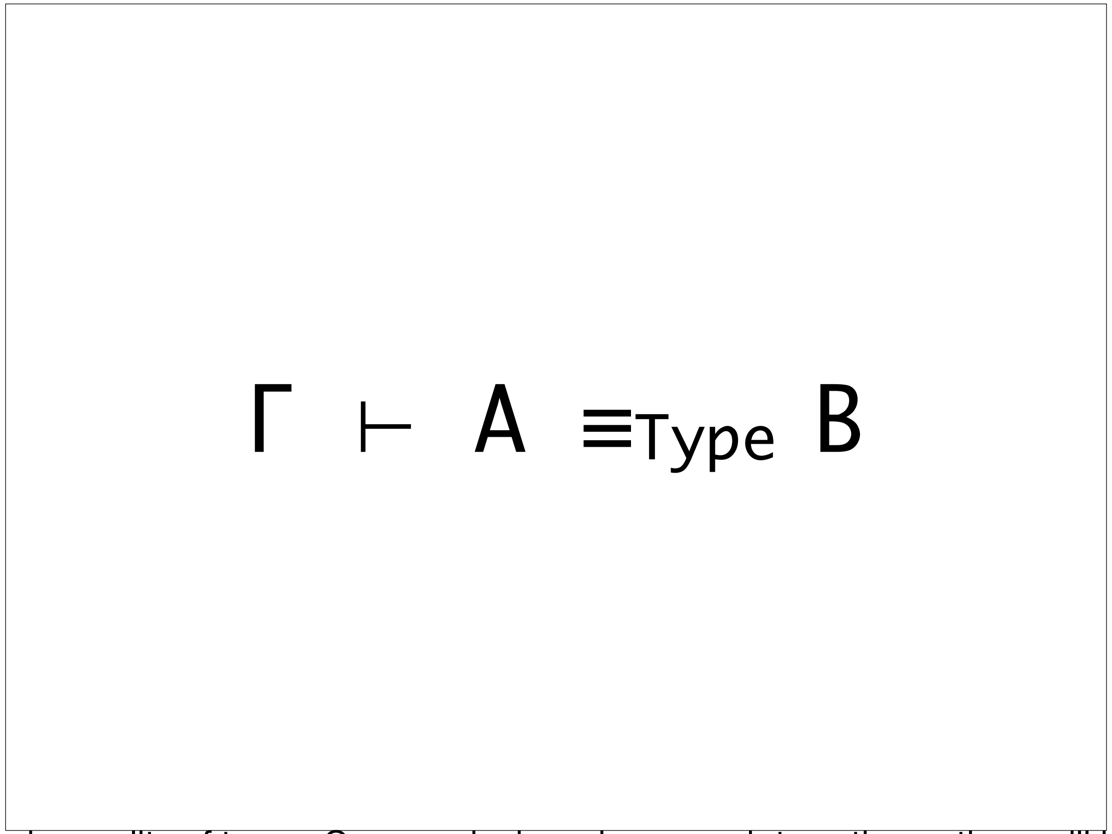
$$\frac{c_1 \to (\Gamma \vdash e_1 : A_1) \qquad \boxed{\Gamma \vdash A_1 \equiv_{\mathrm{Type}} \textstyle\prod x{:}C.D} \qquad c_2 : (\Gamma \vdash D \ \mathrm{type}) \to (\Delta \vdash e_2 : \_)}{c_1 c_2 \to (\boxed{\Gamma \bowtie \Delta} \vdash e_1{}^{@(x:C.D)} e_2 : D[e_2/x])}$$

how to combine contexts?

An example on how the two modes are combined.

There are some problems. First of all, how do we make sure that $A_1$ is a product? In nice type theories the answer would be: normalize $A_1$ and see if you get a product. But we cannot do that here because our type theory is *not* normalizing. In fact, **recognizing the shape of a type is a fundamental activity** that is performed by mathematicians. It is not captured by traditional rules of type theory.

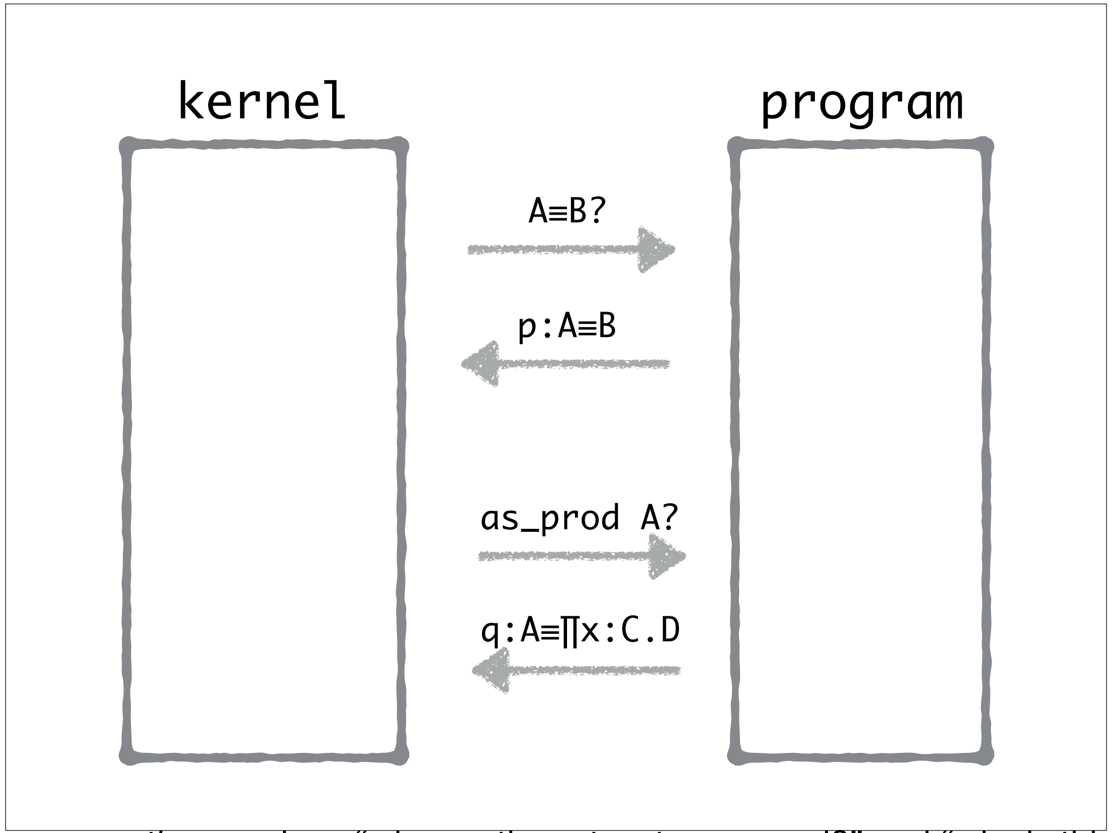
And we need to combine the different contexts, which may not actually be compatible with each other.



$$\Gamma \vdash A \equiv_{\mathsf{Type}} B$$

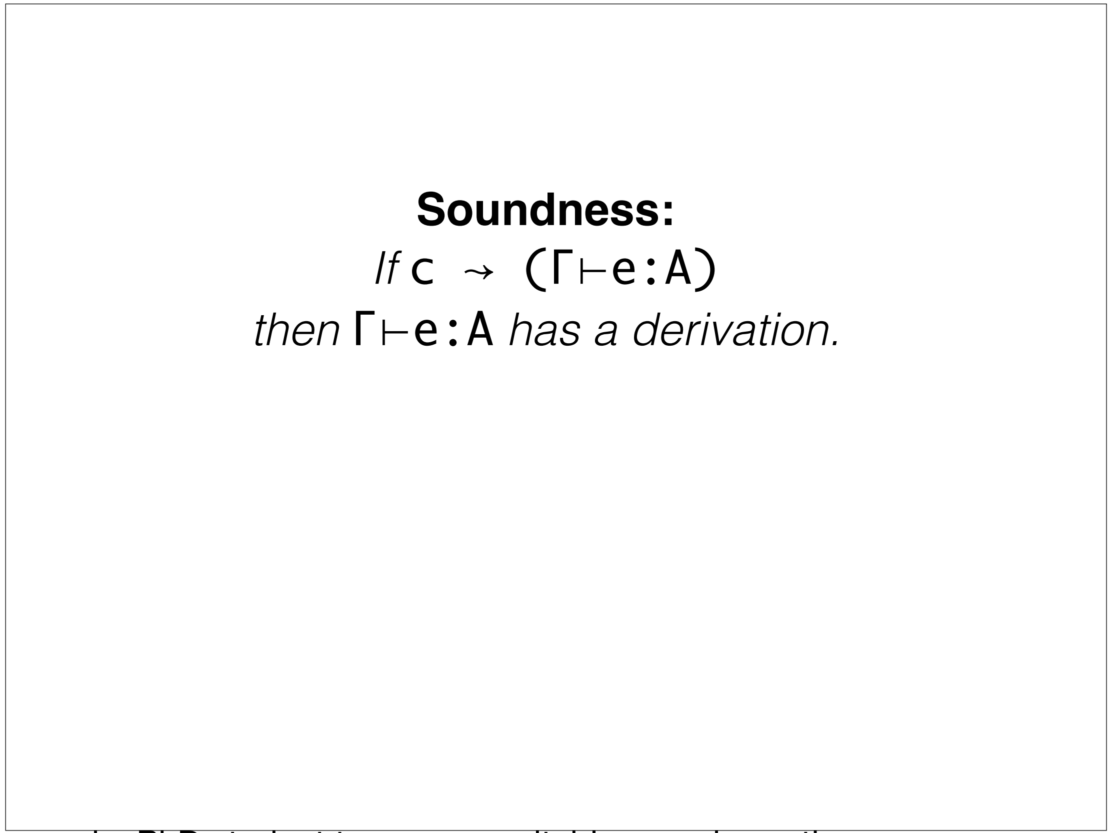
The kernel should be able to check equality of types. Once again, in a nice enough type theory there will be a decision procedure that the kernel can use. This is good because it takes away the burden from the user, and it is bad because it builds in a particular algorithm.

In any case, we have no such algorithm, so the kernel **needs help** in proving equalities.



During evaluation the kernel triggers *questions* such as “why are these two types equal?” and “why is this type a product?” The program *intercepts* the questions and provides answers. The answers are of course computed using the kernel itself, again.

The programming mechanism used to do this is known as *algebraic effects and handlers.* We do not have the time to go into the theory of these. Handlers are a generalization of exception handlers.



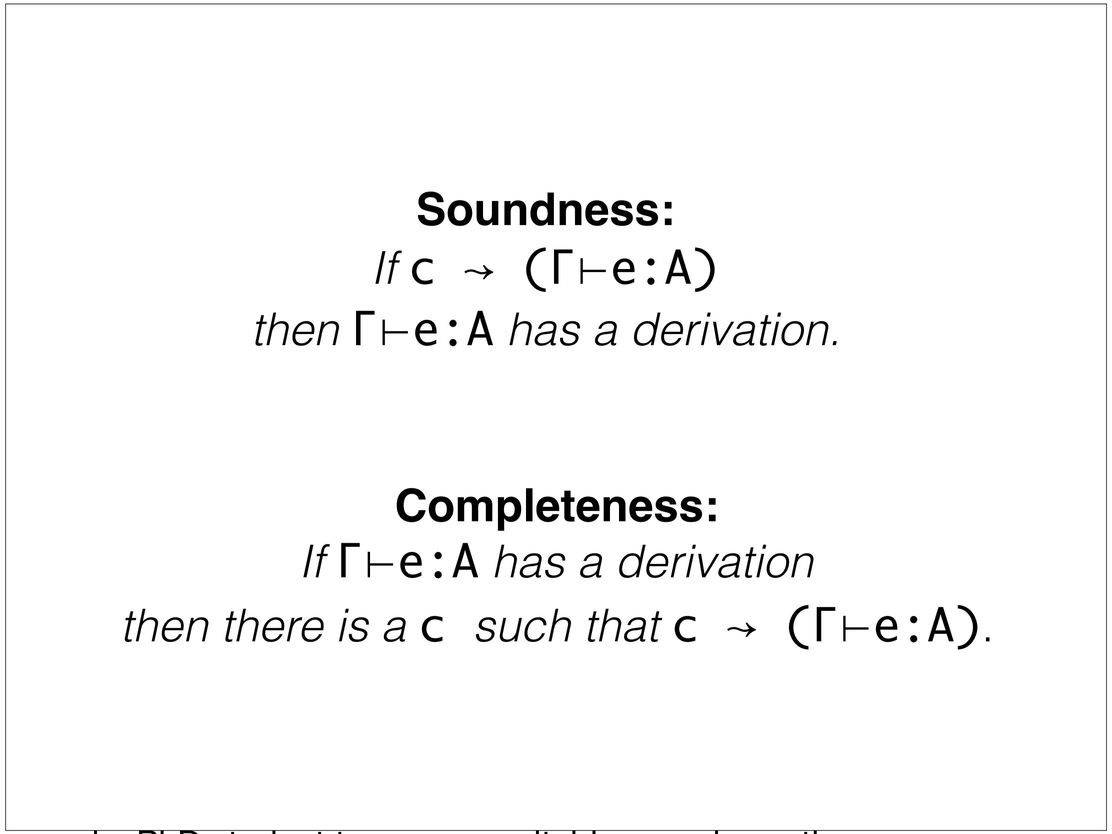
**Soundness:**

*If* $\texttt{c} \rightarrow (\Gamma \vdash \texttt{e:A})$

*then* $\Gamma \vdash \texttt{e:A}$ *has a derivation.*

How can we trust the kernel? We need a PhD student to prove a suitable soundness theorem.

For a solid dissertation the student should also prove a completeness theorem.



**Soundness:**

*If* $\mathtt{c} \rightarrow (\Gamma \vdash \mathtt{e{:}A})$

*then* $\Gamma \vdash \mathtt{e{:}A}$ *has a derivation.*

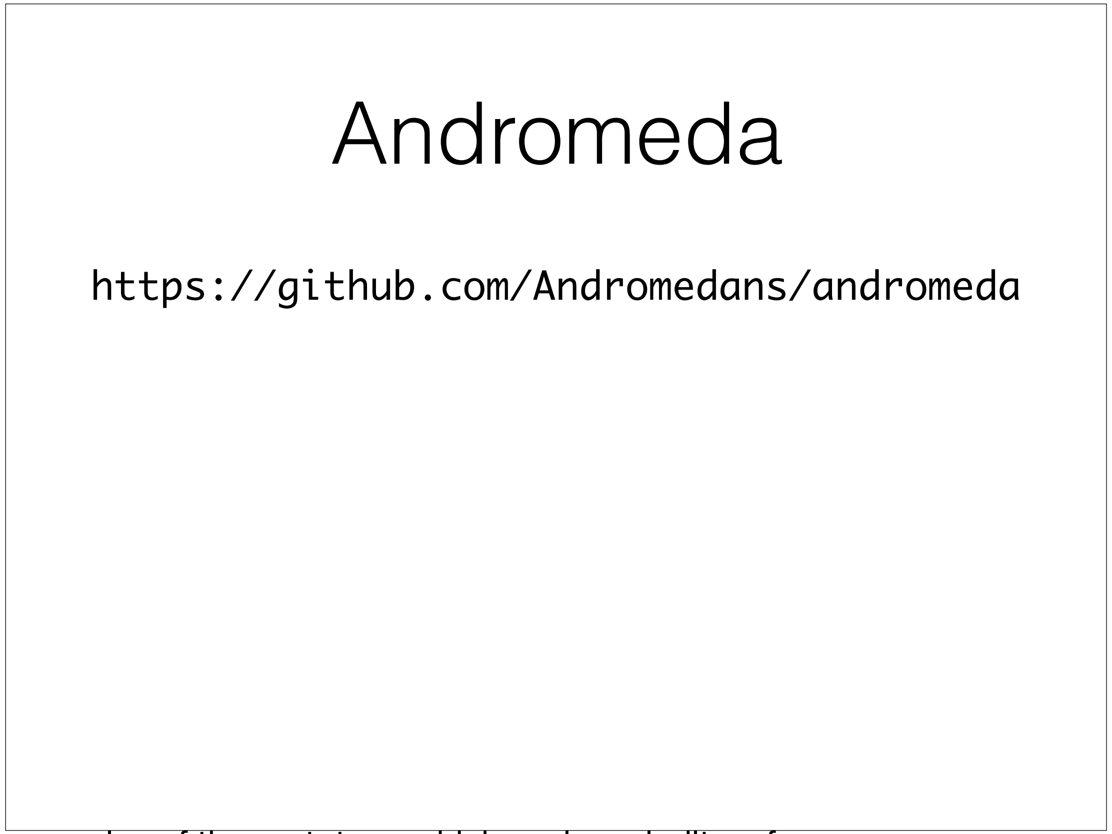
**Completeness:**

*If* $\Gamma \vdash \mathtt{e{:}A}$ *has a derivation*

*then there is a* $\mathtt{c}$ *such that* $\mathtt{c} \rightarrow (\Gamma \vdash \mathtt{e{:}A})$.

How can we trust the kernel? We need a PhD student to prove a suitable soundness theorem.

For a solid dissertation the student should also prove a completeness theorem.



## Andromeda

`https://github.com/Andromedans/andromeda`

Let us finish by looking at some examples of the prototype which we have built so far.

It is implemented in OCaml. The kernel is about 4000 lines of code. We have tried many variants before we started to settle on a system that is very rudimentary. I would say it is most similar to HOL/light, except that it handles dependent types and it mixes programs and terms freely (in HOL/light one cannot embed a program inside a term).

## End of Part III

Thank you for your attention!

This material is based upon work supported by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF under Award No. FA9550-14-1-0096.

Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the Air Force Office of Scientific Research, Air Force Materiel Command, USAF.