The troublesome reflection rule

Andrej Bauer
University of Ljubljana

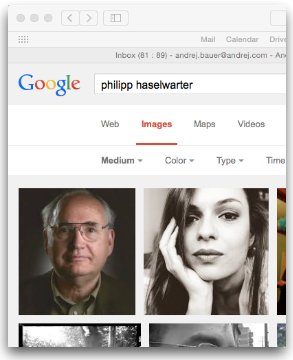
TYPES 2015
Tallinn, May 18–21, 2015

I thank you very much for the invitation. I am a bit worried about talking about type theory at TYPES since I do not consider myself to be a “true” type theorist.

Matija Pretnar

Chris Stone

The work I am presenting is done jointly with Chris Stone from Harvey Mudd College and Matija Pretnar from University of Ljubljana. I like to put up pictures of my coauthors on the slides.
Recently Phillip Haselwarter joined our small team as a PhD student. So I thought I’d find a picture of him on Google.



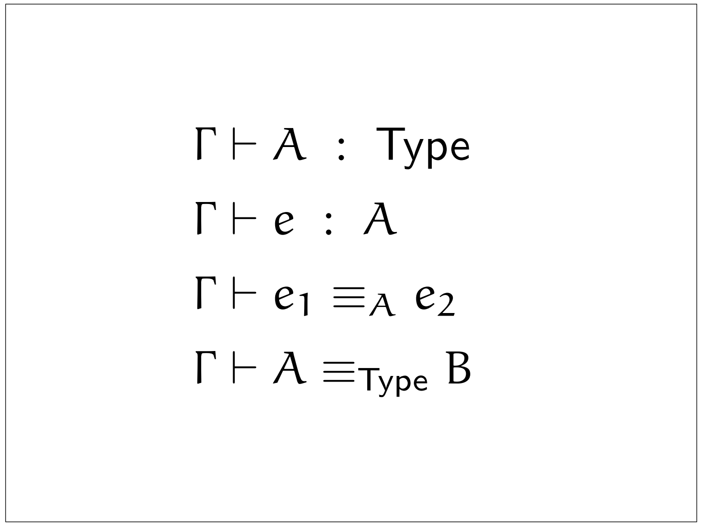
Philipp Haselwarter?

But I got strange results. I know the gentleman on the left and it's not Phillip. Luckily, Phillip is here with us, and you can ask him later to explain these pictures.

Talk outline

- Equality reflection – bad & good
- Current development
- Future possibilities

I am first going to review the equality reflection rule, explain that it is pretty tricky, and argue that we want it anyhow. Then I'll speak about a prototype implementation – which at the moment is called Andromeda – and the challenges that need to be overcome to get something working. I will conclude with general observations that go beyond the reflection rule.



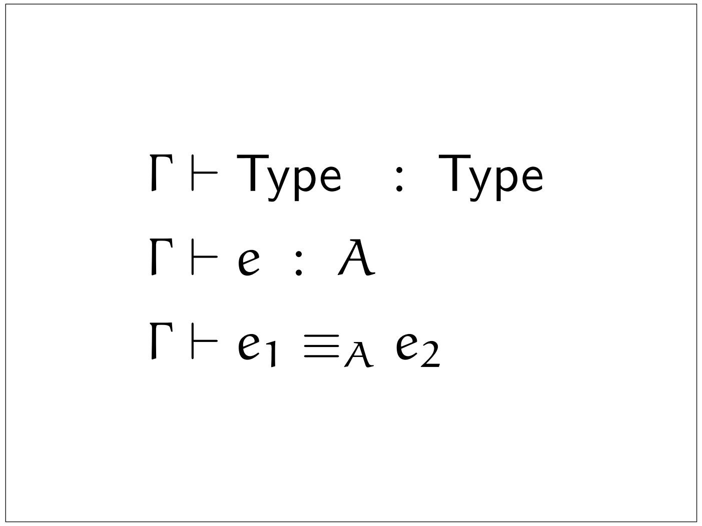
$$\Gamma \vdash A \ : \ \mathsf{Type}$$

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

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

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

We consider traditional type theory with the usual judgments. (I am skipping the judgement that a context is well-formed.) We can reduce these to two judgments by using universes.



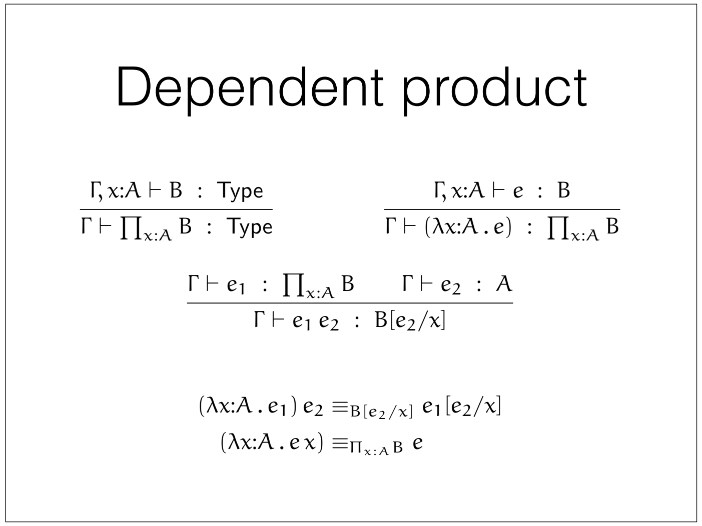
$$\Gamma \vdash \mathsf{Type} \ : \ \mathsf{Type}$$

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

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

If instantiate the type A with Type to recover the judgments for types.

The universes are important, but I am not going to pay any attention to them here, as the issues raised are largely orthogonal to what I would like to discuss. The current implementation of Andromeda actually thinks that Type is in Type, just so that it is easy to play with examples.



Dependent product

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

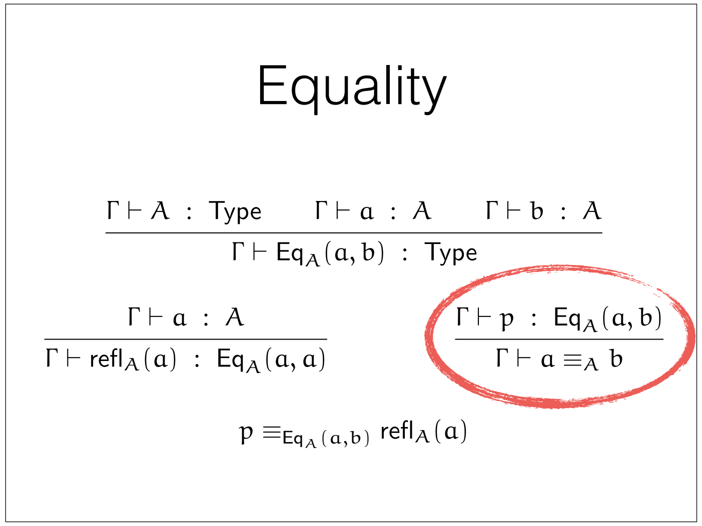
$$\frac{\Gamma \vdash e_1 \;:\; \prod_{x:A} B \qquad \Gamma \vdash e_2 \;:\; A}{\Gamma \vdash e_1\, e_2 \;:\; B[e_2/x]}$$

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

The theory is minimalistic: it has only dependent products and equality types.

The dependent products shown here are standard.

Note that we have the η-rule as well (but no function extensionality).



Equality

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

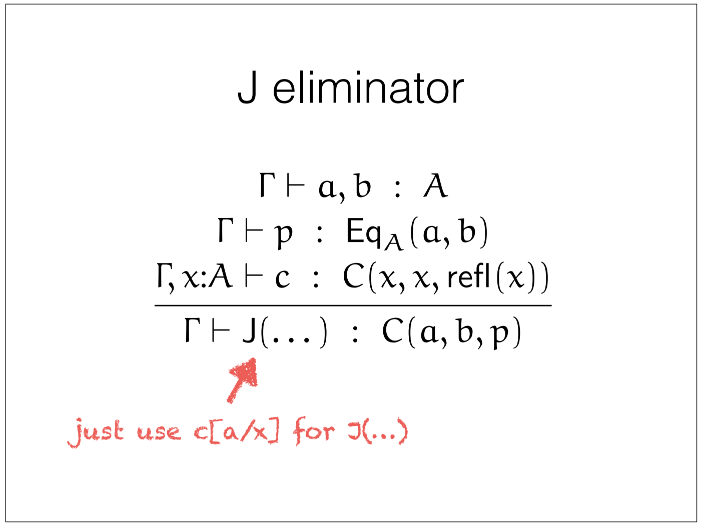
$$\frac{\Gamma \vdash a \;:\; A}{\Gamma \vdash \mathsf{refl}_A(a) \;:\; \mathsf{Eq}_A(a, a)} \qquad \boxed{\frac{\Gamma \vdash p \;:\; \mathsf{Eq}_A(a, b)}{\Gamma \vdash a \equiv_A b}}$$

$$p \equiv_{\mathsf{Eq}_A(a,b)} \mathsf{refl}_A(a)$$

The other type former is equality. Let me call it "equality" rather than "identity" – to keep it distinct from the usual identity type.

The equality type has the expected type former, the reflexivity introduction rule, and "uniqueness-of-equality" equation. It has an unusual elimination rule, which says that the equality type **reflects** judgmental equality.

You must have heard the mantra that "the reflection rule is bad because it breaks decidability of type checking". But let us look more deeply into what is going on.



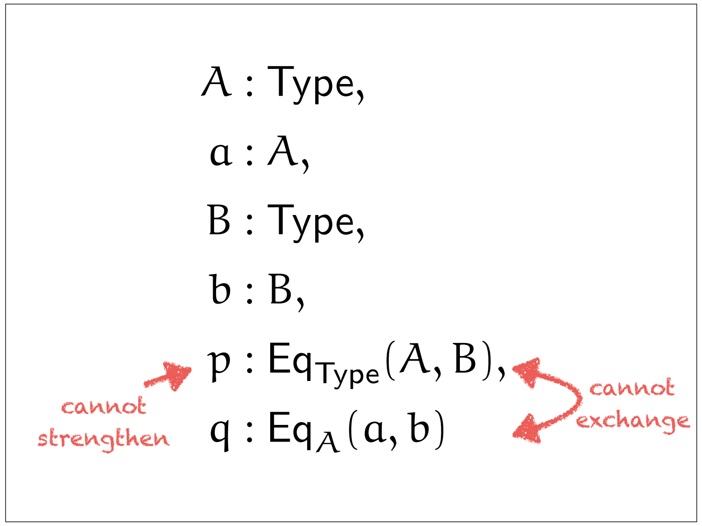
J eliminator

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

just use c[a/x] for J(...)

First, the usual J-eliminator is derivable and it has a very strong conversion rule – because in place of "J(...)" we can simply put "c[a/x]". So in a sense there is no need for J at all.

This is not so interesting. The power of the reflection rule comes from the ability to *hypothesize* new equalities. Let's see where that leads.

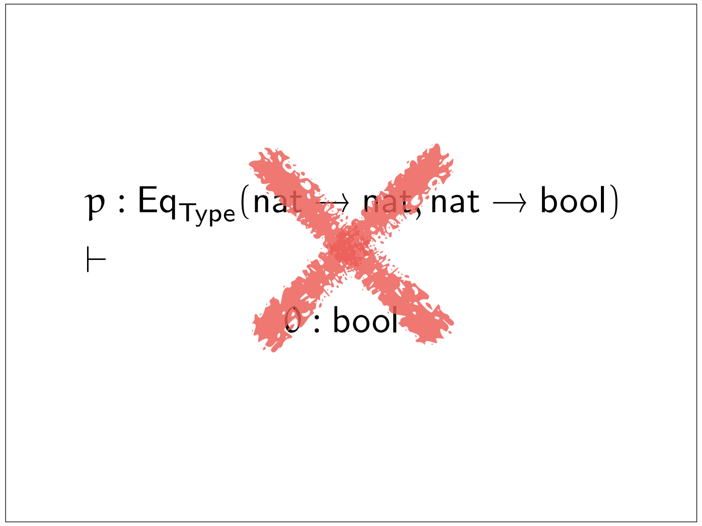


$$\begin{array}{l} A : \mathsf{Type}, \\ a : A, \\ B : \mathsf{Type}, \\ b : B, \\ p : \mathsf{Eq}_{\mathsf{Type}}(A, B), \\ q : \mathsf{Eq}_A(a, b) \end{array}$$

cannot strengthen

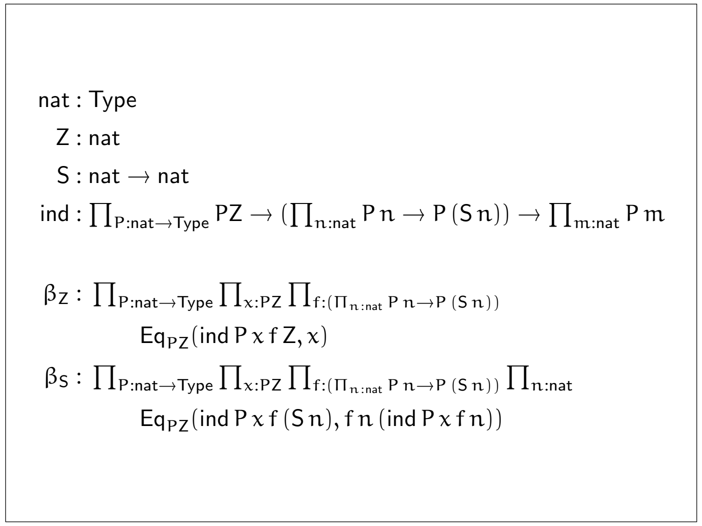
cannot exchange

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.
Is this really a problem? Not as far as standard algorithms are concerned, but is definitely a good source of errors and false intuitions. Perhaps we need fancier contexts to capture dependencies, such as directed acyclic graphs. So far we have gotten away with contexts as lists.



$$p : \mathsf{Eq}_{\mathsf{Type}}(\mathsf{nat} \to \mathsf{nat}, \mathsf{nat} \to \mathsf{bool}) \vdash 0 : \mathsf{bool}$$

It gets worse. Suppose we hypothesize equality of the Baire space and the Cantor space. Then we can show that the identity function on natural numbers has type “nat → bool”. Therefore, the identity applied to 0 is a boolean.
By β-reduction it follows that 0 has type bool as well. This is a mess!
By the way, it is consistent to assume that the Baire and Cantor spaces are equal: consider a skeleton of the category of Sets, e.g. von Neumann cardinals. In it, the Baire space and the Cantor space are actually equal because they both have the cardinality of continuum.
The problem is in β-reduction as stated. We need to be more careful and explicitly tag applications with typing information. We will do this in a moment but let us first see what equality reflection is good for.

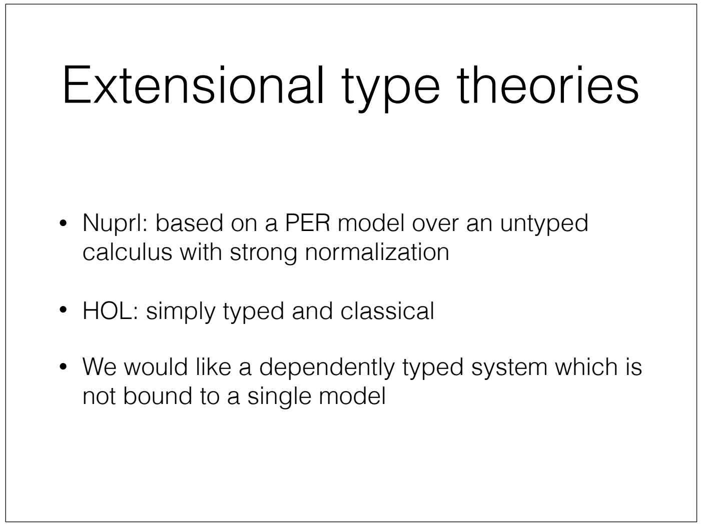


$$\begin{aligned}
&\mathsf{nat} : \mathsf{Type}\\
&\quad \mathsf{Z} : \mathsf{nat}\\
&\quad \mathsf{S} : \mathsf{nat} \to \mathsf{nat}\\
&\mathsf{ind} : \textstyle\prod_{P:\mathsf{nat}\to\mathsf{Type}} P\,\mathsf{Z} \to (\prod_{n:\mathsf{nat}} P\,n \to P\,(\mathsf{S}\,n)) \to \prod_{m:\mathsf{nat}} P\,m
\end{aligned}$$

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

The reflection rule gives us immense expressive power. We can postulate new constructs **and their computation rules.** For instance, by postulating the rules shown we get the natural numbers.
In fact, we can define all standard (and less standard) constructs this way.

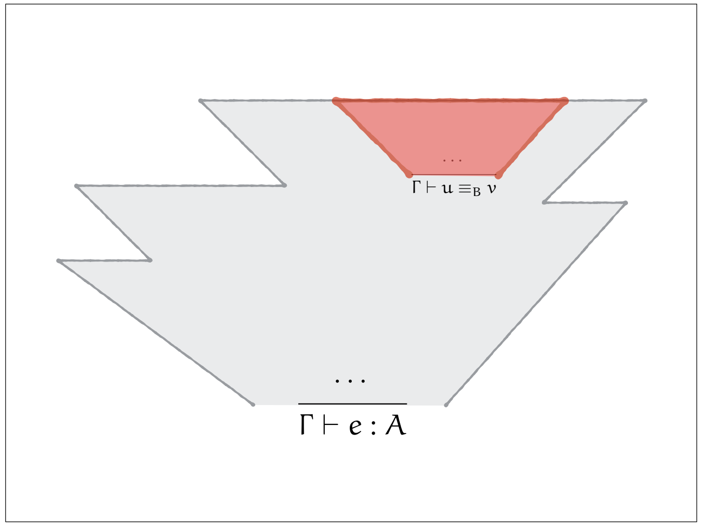
But to make such axiomatizations useful we need a convenient way of using equality reflection.



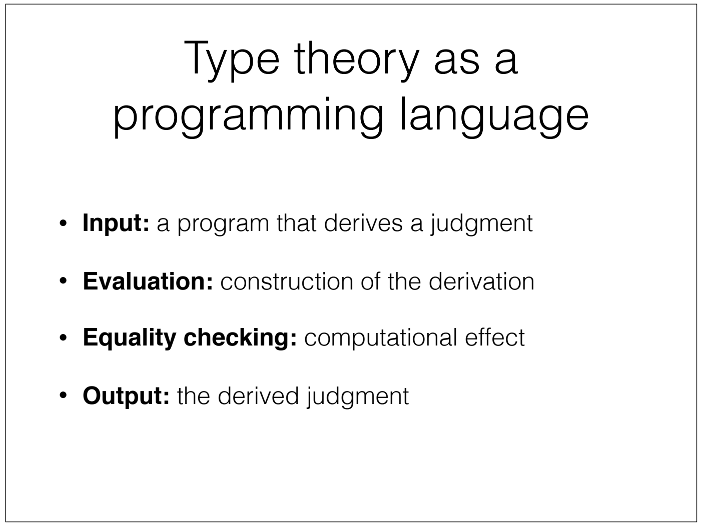
Extensional type theories

- Nuprl: based on a PER model over an untyped calculus with strong normalization
- HOL: simply typed and classical
- We would like a dependently typed system which is not bound to a single model

There are of course other implementations of extensional type theory.
Nuprl is a venerable system that inspired many features of modern proof assistants. Its equality is extensional. It is managed by relying on good computational properties of an underlying untyped calculus.
HOL avoids problems by using simple types and classical logic.
We would like a dependent tyeory which does not rely on a particular interpretation.



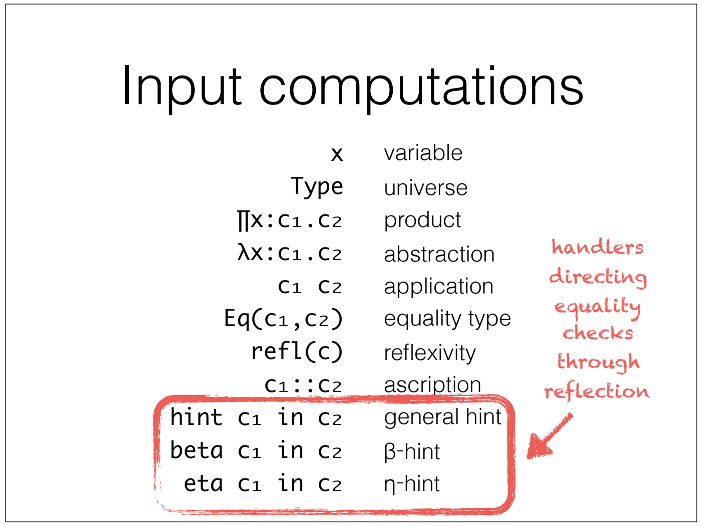
Type checking implicitly generates a derivation tree (shown in gray).
Parts of the derivation tree that are about equality are not recorded in the proof terms (shown red). For a well-behaved type theory the equality derivations can be reconstructed algorithmically.
Our type theory is not well behaved. So we need **advice** on how to check equalities. What form should the advice take?



Type theory as a programming language

- **Input:** a program that derives a judgment
- **Evaluation:** construction of the derivation
- **Equality checking:** computational effect
- **Output:** the derived judgment

We take the view that the input is not raw type theory but rather a **program**.
The evaluation of such a program implicitly constructs a derivation.
Equality checking is a **computational effects**: it does not contribute directly to the output, but it must be executed.
The output is the judgment derived by the execution of the program. **Note well:** the output is insufficient for reconstruction of derivation.

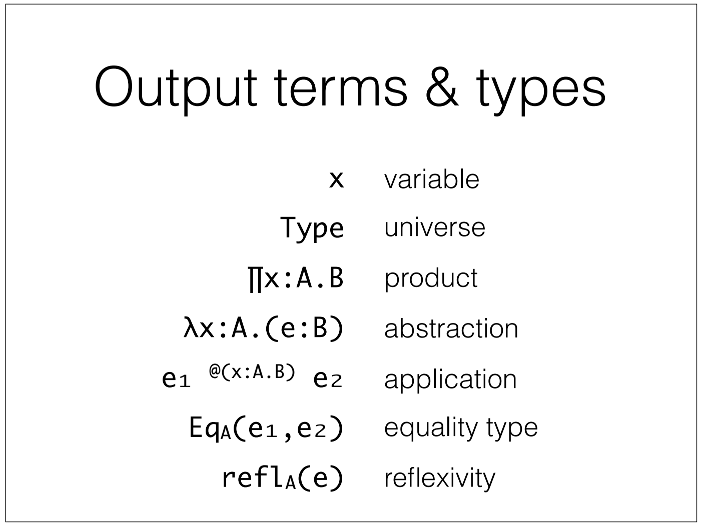


Input computations

`x`	variable
`Type`	universe
`∏x:c₁.c₂`	product
`λx:c₁.c₂`	abstraction
`c₁ c₂`	application
`Eq(c₁,c₂)`	equality type
`refl(c)`	reflexivity
`c₁::c₂`	ascription
`hint c₁ in c₂`	general hint
`beta c₁ in c₂`	β-hint
`eta c₁ in c₂`	η-hint

handlers directing equality checks through reflection

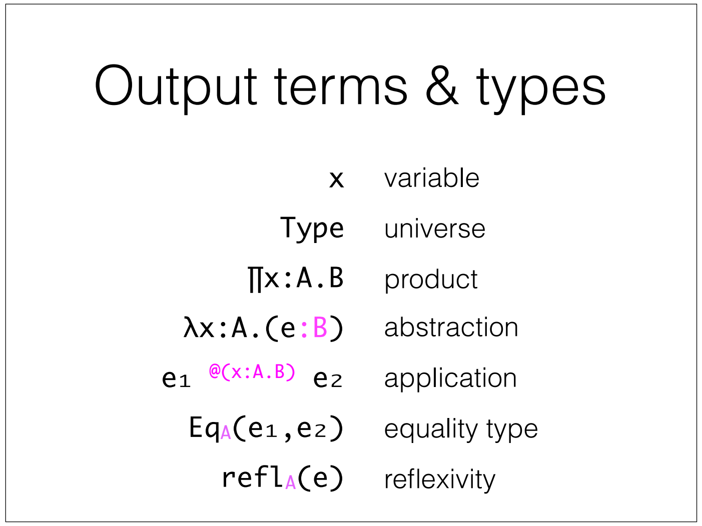
The input computations purposely resemble the terms of type theory, but they are **not** the actual terms. They should be thought of as computations that must be evaluated.
The equality hints are a control mechanism, i.e., **handlers** that direct equality checking. I will explain them later.



Output terms & types

`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

The output terms and types have explicit typing information.



Output terms & types

`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

It is colored magenta.

β-rule

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

$$(\lambda x{:}nat.x{:}nat)\ ^{@(nat\to bool)}\ 0 \not\equiv 0$$

With explicit typing annotations we can write down a sound β-rule.
Our previous example does not reduce because the typing annotations fail to match.

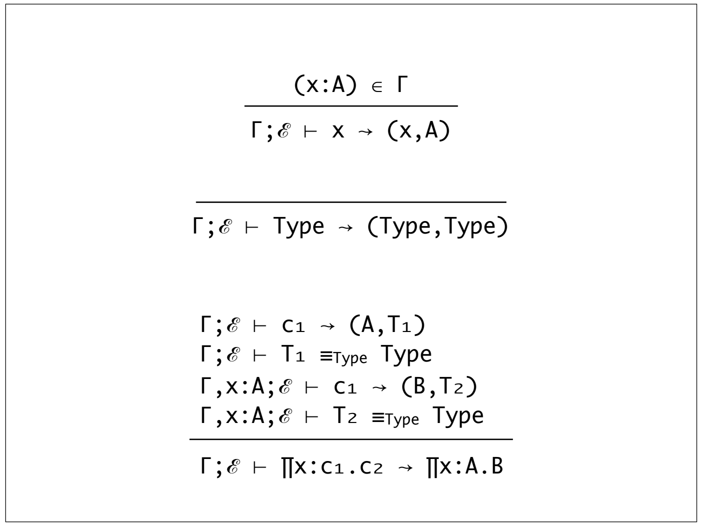
Operational semantics

$$\Gamma;\mathscr{E} \vdash c \to (e,A)$$

“In context Γ using hints $\mathscr{E}$
computation c evaluates to (e,A)”

Soundness:
If $\Gamma;\mathscr{E} \vdash c \to (e,A)$ then $\Gamma \vdash e{:}A$.

We need to give an operational semantics to the programs.
Note that we are evaluating open terms in a typing context Γ.
In addition to the typing context there are also **equality hints** $\mathscr{E}$. They are used by equality checking.
The soundness guarantee is that programs evaluate to derivable judgments. Evaluation could be blocked, or diverge, but it cannot result in an underivable judgment.
Peter Lumsdaine remarked that the input c and the output e seem to have the same structure, i.e., e is essentially c with additional typing information. It would be interesting to formulate a precise claim and prove it.

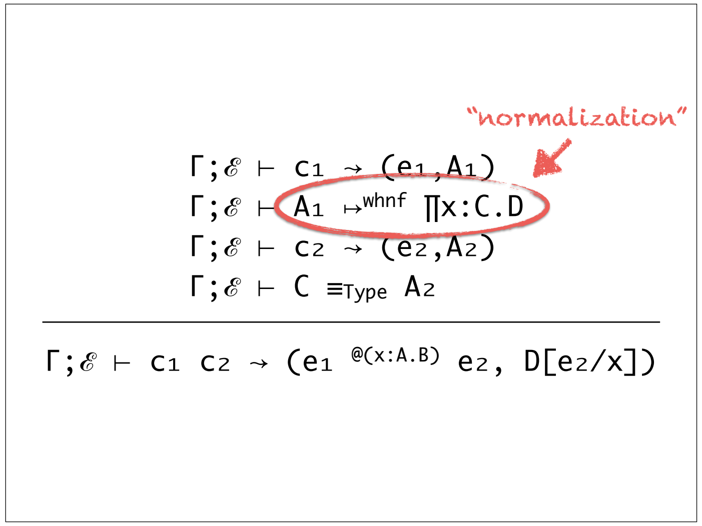


$$\frac{(x{:}A) \in \Gamma}{\Gamma;\mathscr{E} \vdash x \to (x,A)}$$

$$\frac{}{\Gamma;\mathscr{E} \vdash \mathsf{Type} \to (\mathsf{Type},\mathsf{Type})}$$

$$\frac{\begin{array}{l}\Gamma;\mathscr{E} \vdash c_1 \to (A,T_1)\\ \Gamma;\mathscr{E} \vdash T_1 \equiv_{\mathsf{Type}} \mathsf{Type}\\ \Gamma,x{:}A;\mathscr{E} \vdash c_1 \to (B,T_2)\\ \Gamma,x{:}A;\mathscr{E} \vdash T_2 \equiv_{\mathsf{Type}} \mathsf{Type}\end{array}}{\Gamma;\mathscr{E} \vdash \prod x{:}c_1.c_2 \to \prod x{:}A.B}$$

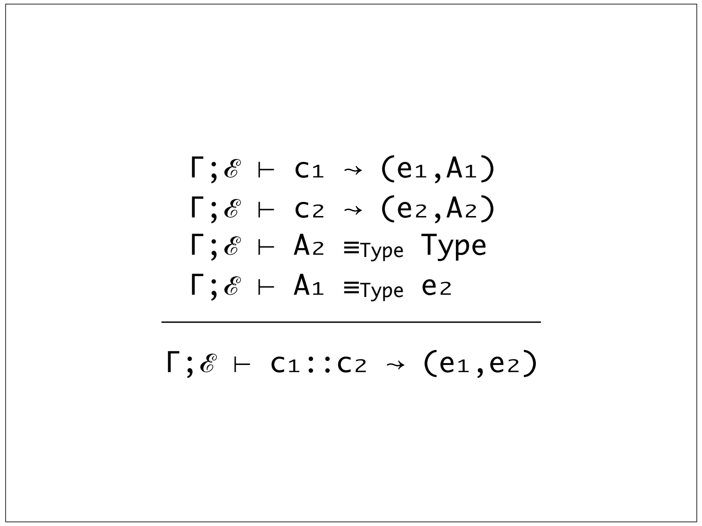
Let us have a look at some of the rules for operational semantics.
The first two are hopefully self-explanatory.
The rule for products shows how equality checks are triggered. For instance, to check that c_1 evaluates to a type we check that its type is Type.



"normalization"

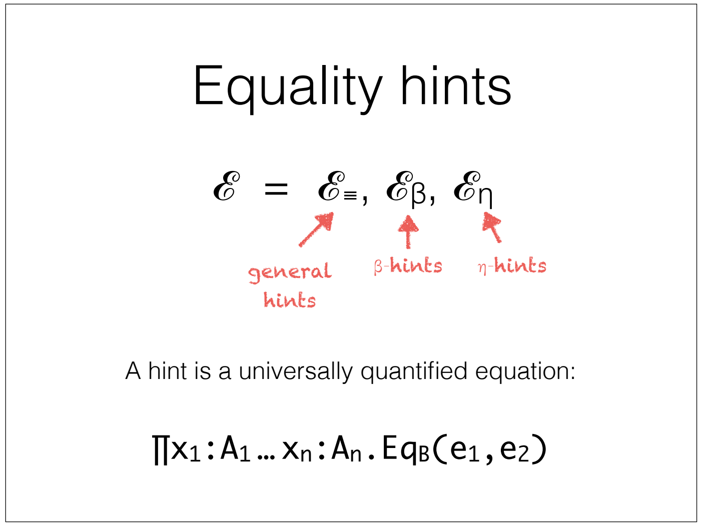
$$\frac{\begin{array}{l}\Gamma;\mathscr{E} \vdash c_1 \to (e_1,A_1)\\ \Gamma;\mathscr{E} \vdash A_1 \mapsto^{\mathsf{whnf}} \prod x{:}C.D\\ \Gamma;\mathscr{E} \vdash c_2 \to (e_2,A_2)\\ \Gamma;\mathscr{E} \vdash C \equiv_{\mathsf{Type}} A_2\end{array}}{\Gamma;\mathscr{E} \vdash c_1\ c_2 \to (e_1 \ {}^{@(x:A.B)}\ e_2,\ D[e_2/x])}$$

The application rule is instructive. First, notice how it inserts the typing annotations into the output.
The rule has to establish the fact that A_1 is a product type. I believe this should be a separate judgment in type theory, namely the act of recognizing the shape of a term or a type.
We just use weak head normalization to discern the structure of the type.
However, it is not really correct to call this "normalization", as the user may install hints that make it diverge.



$$\frac{\begin{array}{l}\Gamma;\mathscr{E} \vdash c_1 \to (e_1,A_1)\\ \Gamma;\mathscr{E} \vdash c_2 \to (e_2,A_2)\\ \Gamma;\mathscr{E} \vdash A_2 \equiv_{\mathrm{Type}} \mathrm{Type}\\ \Gamma;\mathscr{E} \vdash A_1 \equiv_{\mathrm{Type}} e_2\end{array}}{\Gamma;\mathscr{E} \vdash c_1::c_2 \to (e_1,e_2)}$$

Type ascriptions allows the program to control the output type. It is the computational form of type conversion (equal types may be interchanged).



Equality hints

$$\mathscr{E} \;=\; \mathscr{E}_{\equiv},\ \mathscr{E}_{\beta},\ \mathscr{E}_{\eta}$$

general hints

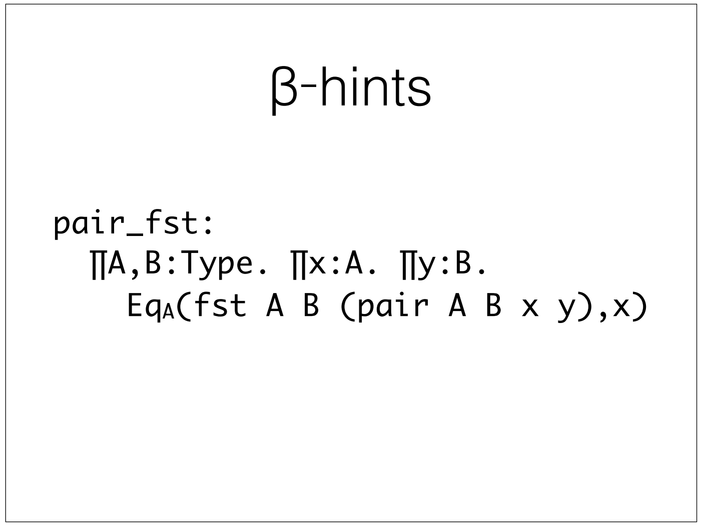
β-hints

η-hints

A hint is a universally quantified equation:

$$\prod x_1:A_1 \ldots x_n:A_n.\,\mathrm{Eq}_B(e_1,e_2)$$

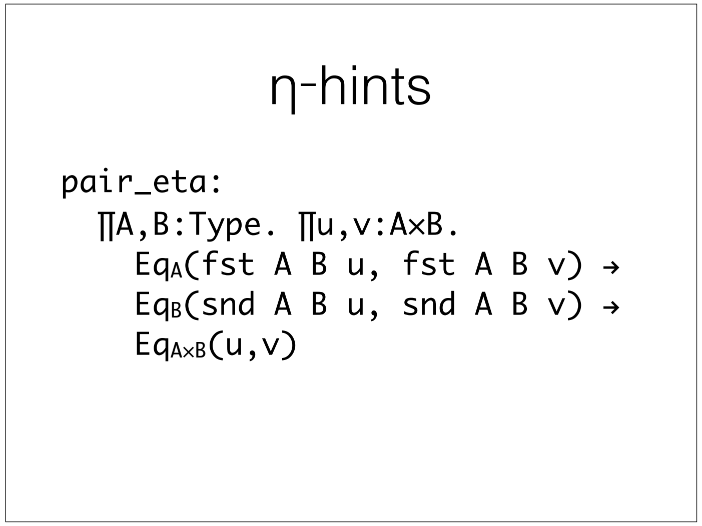
We still have to explain how equality hints work.
At the moment we have three kinds of equality hints: general, beta, and eta hints.
They are used during different phases of equality checking.
A hint is just a universally quantified equation.



β-hints

```
pair_fst:
  ∏A,B:Type. ∏x:A. ∏y:B.
    Eq_A(fst A B (pair A B x y),x)
```

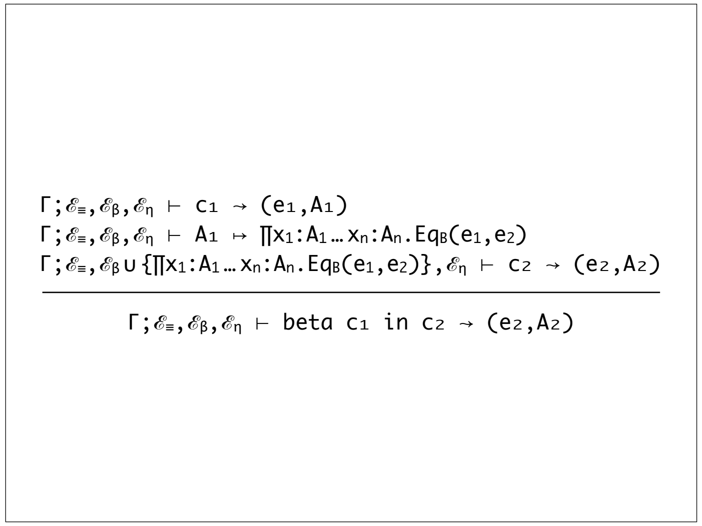
For instance, here is a β-hint that one would use in the axiomatization of simple products.
A β-hint is only useful if it has a certain form: the left-hand side of equality must mention all the quantified variables. This is so because during normalization, when such hints are used, that is the only information available.



η-hints

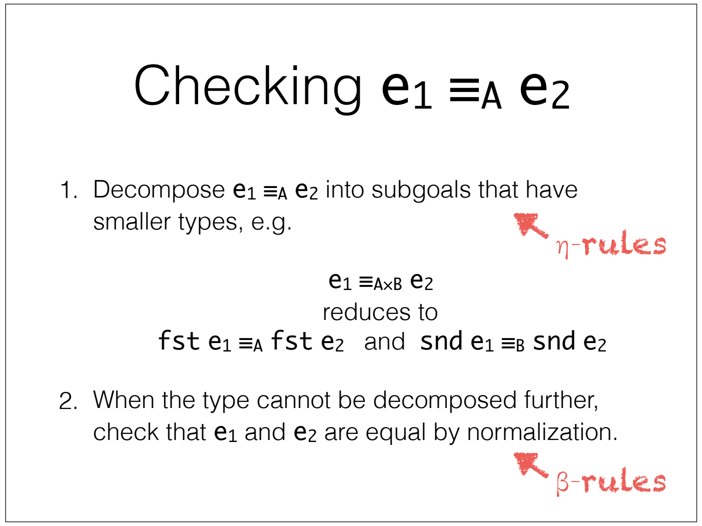
```
pair_eta:
  ∏A,B:Type. ∏u,v:A×B.
    Eq_A(fst A B u, fst A B v) →
    Eq_B(snd A B u, snd A B v) →
    Eq_A×B(u,v)
```

Here is a typical η-hint, again for simple products. Now the shape of the hints is different: the goal may depend on equality proofs which do not appear in the goal, and so they are considered to be new subgoals to be solved recursively. In general we allow universally quantified equality subgoals.



$$\frac{\begin{array}{c}\Gamma;\mathscr{E}_{\equiv},\mathscr{E}_{\beta},\mathscr{E}_{\eta} \vdash \mathtt{c}_1 \to (\mathtt{e}_1,\mathtt{A}_1) \\ \Gamma;\mathscr{E}_{\equiv},\mathscr{E}_{\beta},\mathscr{E}_{\eta} \vdash \mathtt{A}_1 \mapsto \prod \mathtt{x}_1{:}\mathtt{A}_1 \ldots \mathtt{x}_n{:}\mathtt{A}_n.\mathtt{Eq}_{\mathtt{B}}(\mathtt{e}_1,\mathtt{e}_2) \\ \Gamma;\mathscr{E}_{\equiv},\mathscr{E}_{\beta} \cup \{\prod \mathtt{x}_1{:}\mathtt{A}_1 \ldots \mathtt{x}_n{:}\mathtt{A}_n.\mathtt{Eq}_{\mathtt{B}}(\mathtt{e}_1,\mathtt{e}_2)\},\mathscr{E}_{\eta} \vdash \mathtt{c}_2 \to (\mathtt{e}_2,\mathtt{A}_2)\end{array}}{\Gamma;\mathscr{E}_{\equiv},\mathscr{E}_{\beta},\mathscr{E}_{\eta} \vdash \mathtt{beta}\ \mathtt{c}_1\ \mathtt{in}\ \mathtt{c}_2 \to (\mathtt{e}_2,\mathtt{A}_2)}$$

The evaluation of beta hints just installs the hint and proceeds with evaluation.



Checking $e_1 \equiv_A e_2$

1. Decompose $e_1 \equiv_A e_2$ into subgoals that have smaller types, e.g.

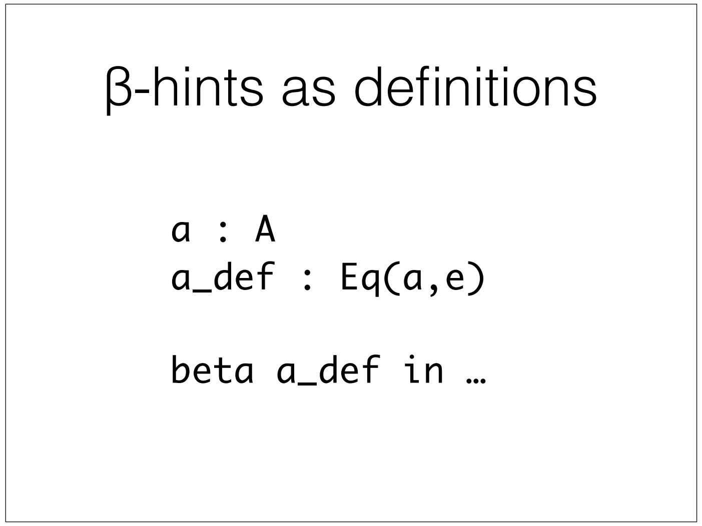
   $e_1 \equiv_{A\times B} e_2$
   reduces to
   $\mathtt{fst}\ e_1 \equiv_A \mathtt{fst}\ e_2$ and $\mathtt{snd}\ e_1 \equiv_B \mathtt{snd}\ e_2$

   ↖ η-rules

2. When the type cannot be decomposed further, check that e_1 and e_2 are equal by normalization.

   ↖ β-rules

We still need an algorithmic way of “checking” equality. We use type-directed checking a la Harper & Stone. There are two phases.
The first phase decomposes an equation into subgoals with smaller types.
When the type cannot be decomposed anymore, a normalization phase checks whether the terms are equal.
The two phases use the β- and η-rules, respectively. They also consult the respective hints. It is easy to break termination of the procedure by installing silly hints.
General hints are consulted **before** the first phase. With these we can resolve goals that do not have the shape of β- and η-rules, such as commutativity and associativity of addition.

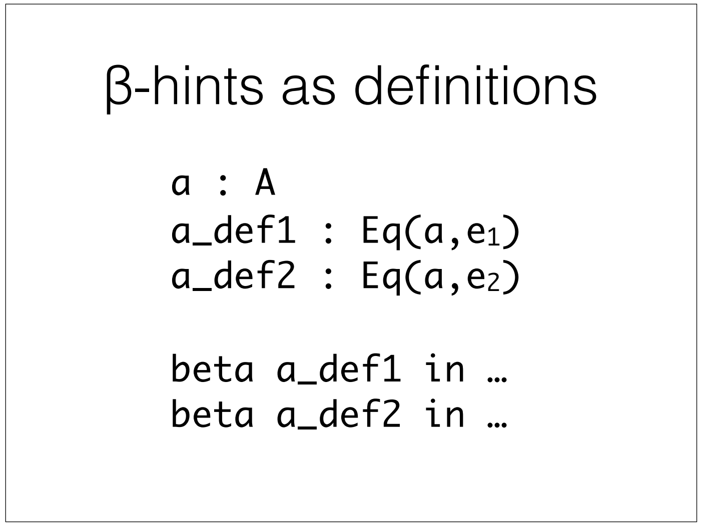


β-hints as definitions

```
a : A
a_def : Eq(a,e)

beta a_def in …
```

When playing even with just the rudimentary prototype we have now, one quickly gets ideas.
For example, the β-hints can be used as definitions.
We can postulate the existence of an element a of type A, and the fact that it is equal to an expression e. Then, if we want a to be automatically unfolded into e, we just use the beta hint a_def.
But we do not have to, so we can also leave a unchanged.



β-hints as definitions

```
a : A
a_def1 : Eq(a,e_1)
a_def2 : Eq(a,e_2)

beta a_def1 in …
beta a_def2 in …
```

Nothing forces us to have just one definition of a! It may be convenient to have two, or any number of them, and then we control their use with β-hints.

I think this is just the tip of the iceberg.

What else?

- Voevodsky's Homotopy Type System
- Enrich the input language with other computational effects and handlers
- Implement standard proof assistant techniques (implicit arguments, proof search, type clases, …) in the language

There are still many things to do. We are currently cleaning up the implementation and should soon have a system which follows the ideas presented here.
But then we plan to go on. One goal is to implement Voevodsky's HTS, a type system with a notion of fibered type and two types of equality. This should have HoTT applications. Another is to extend the input language with a richer set of computational effects that allow us to implement the techniques found in proof assistants: implicit arguments, proof search, unification, type classes, etc. Thee should all be definable in our language – but it will take a smart PhD student to do it.
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.