Mtac: A Monad for Typed Tactic Programming in Coq

Beta Ziliani

Max Planck Institute for Software Systems
(MPI-SWS)

joint work with Derek Dreyer (MPI-SWS),
Neel Krishnaswami (MPI-SWS), Aleks Nanevski (IMDEA)
and Viktor Vafeiadis (MPI-SWS)

July 5, 2014

A simple problem in Coq

$$\frac{x\ y\ z{:}A \qquad s{:}\mathsf{list}\ A}{x \in \mathsf{append}\ s\ (z :: x :: y :: [])}$$

A simple problem in Coq

$$\frac{x\ y\ z{:}A \qquad s{:}\mathsf{list}\ A}{x \in \mathsf{append}\ s\ (z :: x :: y :: [])}$$

Proof.

A simple problem in Coq

$$\text{inR} : a \in r \to a \in \text{append } l\ r$$

$$\frac{x\ y\ z{:}A \qquad s{:}\text{list } A}{x \in \text{append } s\ (z :: x :: y :: [])}$$

Proof.

apply: inR.

A simple problem in Coq

$$\text{inR} : a \in r \to a \in \text{append } l\ r$$

$$\frac{x\ y\ z{:}A \qquad s{:}\text{list } A}{x \in (z :: x :: y :: [])}$$

Proof.

apply: inR.

A simple problem in Coq

$$\frac{x\ y\ z{:}A \qquad s{:}\mathsf{list}\ A}{x \in (z :: x :: y :: [])}$$

Proof.

apply: inR.

A simple problem in Coq

$$\frac{x\ y\ z{:}A \qquad s{:}\mathsf{list}\ A}{x \in (z :: x :: y :: [])}$$

Proof.

apply: inR.

A simple problem in Coq

$$\text{in\_tail} : a \in l \to a \in (b :: l)$$

$$\frac{x\ y\ z{:}A \qquad s{:}\text{list}\ A}{x \in (z :: x :: y :: [])}$$

Proof.

apply: inR.

apply: in_tail.

A simple problem in Coq

$$\text{in\_tail} : a \in l \to a \in (b :: l)$$

$$\frac{x\ y\ z{:}A \qquad s{:}\text{list}\ A}{x \in (x :: y :: [])}$$

Proof.

apply: inR.

apply: in_tail.

A simple problem in Coq

$$\frac{x\ y\ z{:}A \qquad s{:}\mathsf{list}\ A}{x \in (x :: y :: [])}$$

Proof.

apply: inR.

apply: in_tail.

A simple problem in Coq

$$\text{in\_head} : a \in (a :: l)$$

$$\frac{x\ y\ z{:}A \qquad s{:}\text{list}\ A}{x \in (x :: y :: [])}$$

Proof.

apply: inR.

apply: in_tail.

apply: in_head.

A simple problem in Coq

$$\frac{x\ y\ z{:}A \qquad s{:}\mathsf{list}\ A}{x \in (x :: y :: [])}$$

□

Proof.

apply: inR.

apply: in_tail.

apply: in_head.

Qed.

A simple problem in Coq

This script is:
Boring.

$$\frac{x\ y\ z{:}A \qquad s{:}\mathsf{list}\ A}{x \in (x :: y :: [])}$$

□

Proof.
apply: inR.
apply: in_tail.
apply: in_head.
Qed.

A simple problem in Coq

This script is:

Boring.

Fragile.

$$\frac{x\ y\ z{:}A \qquad s{:}\mathsf{list}\ A}{x \in (x :: y :: [])}$$

□

Proof.

apply: inR.

apply: in_tail.

apply: in_head.

Qed.

A simple problem in Coq

This script is:

Boring.

Fragile.

$$\frac{x\ y\ z{:}A \qquad s{:}\mathsf{list}\ A}{x \in (y :: x :: [])}$$

$\square$

Proof.

apply: inR.

apply: in_tail.

apply: in_head.

Qed.

search in Ltac

```
(1)   Ltac search x s :=
(2)       match s with
(3)       | append ?l ?r ⇒ let p := search x l in
(4)                          constr: (inL r p)
(5)       | append ?l ?r ⇒ let p := search x r in
(6)                          constr: (inR l p)
(7)       | (x :: ?s') ⇒ constr: (in_head x s)
(8)       | (?y :: ?s') ⇒ let p := search x s' in
(9)                        constr: (in_tail y x s' p)
(10)      end.
```

search in Ltac

Logic of Coq includes a funct. language

```
(1)   Ltac search x s :=
(2)       match s with
(3)       | append ?l ?r ⇒ let p := search x l in
(4)                          constr: (inL r p)
(5)       | append ?l ?r ⇒ let p := search x r in
(6)                          constr: (inR l p)
(7)       | (x :: ?s') ⇒ constr: (in_head x s)
(8)       | (?y :: ?s') ⇒ let p := search x s' in
(9)                        constr: (in_tail y x s' p)
(10)      end.
```

search in Ltac

Logic of Coq includes a pure funct. language

```
(1)   Ltac search x s :=
(2)     match s with
(3)     | append ?l ?r ⇒ let p := search x l in
(4)                      constr: (inL r p)
(5)     | append ?l ?r ⇒ let p := search x r in
(6)                      constr: (inR l p)
(7)     | (x :: ?s') ⇒ constr: (in_head x s)
(8)     | (?y :: ?s') ⇒ let p := search x s' in
(9)                      constr: (in_tail y x s' p)
(10)    end.
```

search in Ltac

Logic of Coq includes a pure funct. language

search requires impure features

```
(1)   Ltac search x s :=
(2)     match s with
(3)     | append ?l ?r ⇒ let p := search x l in
(4)                      constr: (inL r p)
(5)     | append ?l ?r ⇒ let p := search x r in
(6)                      constr: (inR l p)
(7)     | (x :: ?s') ⇒ constr: (in_head x s)
(8)     | (?y :: ?s') ⇒ let p := search x s' in
(9)                      constr: (in_tail y x s' p)
(10)    end.
```

search in Ltac


```
(1)   Ltac search x s :=
(2)     match s with
(3)     | append ?l ?r ⇒ let p := search x l in
(4)                      constr: (inL r p)
(5)     | append ?l ?r ⇒ let p := search x r in
(6)                      constr: (inR l p)
(7)     | (x :: ?s') ⇒ constr: (in_head x s)
(8)     | (?y :: ?s') ⇒ let p := search x s' in
(9)                      constr: (in_tail y x s' p)
(10)    end.
```

search in Ltac

#2: Unbounded recursion

```
(1)   Ltac search x s :=
(2)       match s with
(3)       | append ?l ?r ⇒ let p := search x l in
(4)                        constr: (inL r p)
(5)       | append ?l ?r ⇒ let p := search x r in
(6)                        constr: (inR l p)
(7)       | (x :: ?s') ⇒ constr: (in_head x s)
(8)       | (?y :: ?s') ⇒ let p := search x s' in
(9)                        constr: (in_tail y x s' p)
(10)      end.
```

search in Ltac

#3: Control flow (backtracking)

```
(1)   Ltac search x s :=
(2)     match s with
(3)     | append ?l ?r ⇒ let p := search x l in
(4)                      constr: (inL r p)
(5)     | append ?l ?r ⇒ let p := search x r in
(6)                      constr: (inR l p)
(7)     | (x :: ?s') ⇒ constr: (in_head x s)
(8)     | (?y :: ?s') ⇒ let p := search x s' in
(9)                      constr: (in_tail y x s' p)
(10)    end.
```

search in Ltac

Untyped!

```
(1)   Ltac search x s :=
(2)       match s with
(3)       | append ?l ?r ⇒ let p := search x l in
(4)                        constr: (inL r p)
(5)       | append ?l ?r ⇒ let p := search x r in
(6)                        constr: (inR l p)
(7)       | (x :: ?s') ⇒ constr: (in_head x s)
(8)       | (?y :: ?s') ⇒ let p := search x s' in
(9)                        constr: (in_tail y x s' p)
(10)      end.
```

search in Ltac

Untyped!

No spec: search :???

```
(1)   Ltac search x s :=
(2)       match s with
(3)       | append ?l ?r ⇒ let p := search x l in
(4)                          constr: (inL r p)
(5)       | append ?l ?r ⇒ let p := search x r in
(6)                          constr: (inR l p)
(7)       | (x :: ?s') ⇒ constr: (in_head x s)
(8)       | (?y :: ?s') ⇒ let p := search x s' in
(9)                        constr: (in_tail y x s' p)
(10)      end.
```

search in Ltac

Untyped!

Hard to maintain

```
(1)   Ltac search x s :=
(2)       match s with
(3)       | append ?l ?r ⇒ let p := search x l in
(4)                          constr: (inL r p)
(5)       | append ?l ?r ⇒ let p := search x r in
(6)                          constr: (inR l p)
(7)       | (x :: ?s') ⇒ constr: (in_head x s)
(8)       | (?y :: ?s') ⇒ let p := search x s' in
(9)                        constr: (in_tail y x s' p)
(10)      end.
```

search in Ltac

Untyped!

Hard to maintain

```
(1)   Ltac search x s :=
(2)       match s with
(3)       | append ?l ?r ⇒ let p := search x l in
(4)                          constr: (inL r p)
(5)       | append ?l ?r ⇒ let p := search x r in
(6)                          constr: (inR l p)
(7)       | (x :: ?s') ⇒ constr: (in_head x s)
(8)       | (?y :: ?s') ⇒ let p := search x s' in
(9)                        constr: (in_tail y x s' p)
(10)      end.
```

search in Ltac

Untyped!

Hard to maintain

```
(1)   Ltac search x s :=
(2)       match s with
(3)       | append ?l ?r ⇒ let p := search x l in
(4)                        constr: (inL r p)
(5)       | append ?l ?r ⇒ let p := search x r in
(6)                        constr: (inR l p)
(7)       | (x :: ?s') ⇒ constr: (in_head x s) : x ∈ x :: s
(8)       | (?y :: ?s') ⇒ let p := search x s' in
(9)                        constr: (in_tail y x s' p)
(10)      end.
```

search in Ltac

Untyped!

Debugging hell!

```
(1)   Ltac search x s :=
(2)       match s with
(3)       | append ?l ?r ⇒ let p := search x l in
(4)                          constr: (inL r p)
(5)       | append ?l ?r ⇒ let p := search x r in
(6)                          constr: (inR l p)
(7)       | (x :: ?s') ⇒ constr: (in_head x s) : x ∈ x :: s
(8)       | (?y :: ?s') ⇒ let p := search x s' in
(9)                          constr: (in_tail y x s' p)
(10)      end.
```

Use Monads

Use Monads

1. Encapsulate tactical effects in a monad.

Use Monads

1. Encapsulate tactical effects in a monad.

   Give tactical effects a Coq type M *A*.

Use Monads

1. Encapsulate tactical effects in a monad.

   Give tactical effects a Coq type M *A*.

2. Execute (**run**) tactics at type inference.

Use Monads

1. Encapsulate tactical effects in a monad.

   Give tactical effects a Coq type M *A*.

2. Execute (**run**) tactics at type inference.

} Mtac

search in Mtac

search in Mtac

Specification (type) should be:

$$\text{search} : \forall (x : A)\ (s : \text{list}\ A).\ \text{M}\ (x \in s).$$

search in Mtac

```
(1)  Definition search (x : A) :=
(2)      mfix f (s : list A) : M (x ∈ s) :=
(3)      mmatch s with
(4)      | [l r] append l r ⇒ mtry  p ← f l;
(5)                                 ret (inL r p)
(6)                            with _ ⇒ p ← f r;
(7)                                 ret (inR l p)
(8)      | [s'] (x :: s') ⇒ ret (in_head x s)
(9)      | [y s'] (y :: s') ⇒ p ← f s'; ret (in_tail y _ _ p)
(10)     | _ ⇒ raise NotFound
(11)     end.
```

search in Mtac


```
(1)   Definition search (x : A) :=
(2)     mfix f (s : list A) : M (x ∈ s) :=
(3)     mmatch s with
(4)     | [l r] append l r ⇒ mtry  p ← f l;
(5)                                ret (inL r p)
(6)                           with _ ⇒ p ← f r;
(7)                                ret (inR l p)
(8)     | [s'] (x :: s') ⇒ ret (in_head x s)
(9)     | [y s'] (y :: s') ⇒ p ← f s'; ret (in_tail y _ _ p)
(10)    | _ ⇒ raise NotFound
(11)    end.
```

search in Mtac

#2: Unbounded recursion

```
(1)  Definition search (x : A) :=
(2)    mfix f (s : list A) : M (x ∈ s) :=
(3)      mmatch s with
(4)      | [l r] append l r ⇒ mtry  p ← f l;
(5)                                 ret (inL r p)
(6)                            with _ ⇒ p ← f r;
(7)                                 ret (inR l p)
(8)      | [s'] (x :: s') ⇒ ret (in_head x s)
(9)      | [y s'] (y :: s') ⇒ p ← f s'; ret (in_tail y _ _ p)
(10)     | _ ⇒ raise NotFound
(11)     end.
```

search in Mtac

#3: Control flow (exceptions)

```
(1)  Definition search (x : A) :=
(2)      mfix f (s : list A) : M (x ∈ s) :=
(3)      mmatch s with
(4)      | [l r] append l r ⇒ mtry  p ← f l;
(5)                                 ret (inL r p)
(6)                            with _ ⇒ p ← f r;
(7)                                 ret (inR l p)
(8)      | [s'] (x :: s') ⇒ ret (in_head x s)
(9)      | [y s'] (y :: s') ⇒ p ← f s'; ret (in_tail y _ _ p)
(10)     | _ ⇒ raise NotFound
(11)     end.
```

search in Mtac

```
(1)  Definition search (x : A) :=
(2)      mfix f (s : list A) : M (x ∈ s) :=
(3)      mmatch s with
(4)      | [l r] append l r ⇒ mtry  p ← f l;
(5)                                 ret (inL r p)
(6)                            with _ ⇒ p ← f r;
(7)                                 ret (inR l p)
(8)      | [s'] (x :: s') ⇒ ret (in_head x s)
(9)      | [y s'] (y :: s') ⇒ p ← f s'; ret (in_tail y _ _ p)
(10)     | _ ⇒ raise NotFound
(11)     end.
```

search in Mtac

Typed!

```
(1)   Definition search (x : A) :=
(2)       mfix f (s : list A) : M (x ∈ s) :=
(3)       mmatch s with
(4)       | [l r] append l r ⇒ mtry  p ← f l;
(5)                                  ret (inL r p)
(6)                              with _ ⇒ p ← f r;
(7)                                  ret (inR l p)
(8)       | [s'] (x :: s') ⇒ ret (in_head x s)
(9)       | [y s'] (y :: s') ⇒ p ← f s'; ret (in_tail y _ _ p)
(10)      | _ ⇒ raise NotFound
(11)      end.
```

search in Mtac

Typed!

Spec: $\forall x\ s.\ \mathsf{M}\ (x \in s)$

```
(1)  Definition search (x : A) :=
(2)      mfix f (s : list A) : M (x ∈ s) :=
(3)      mmatch s with
(4)      | [l r] append l r ⇒ mtry  p ← f l;
(5)                                 ret (inL r p)
(6)                            with _ ⇒ p ← f r;
(7)                                 ret (inR l p)
(8)      | [s'] (x :: s') ⇒ ret (in_head x s)
(9)      | [y s'] (y :: s') ⇒ p ← f s'; ret (in_tail y _ _ p)
(10)     | _ ⇒ raise NotFound
(11)     end.
```

search in Mtac

Typed!

Type checker

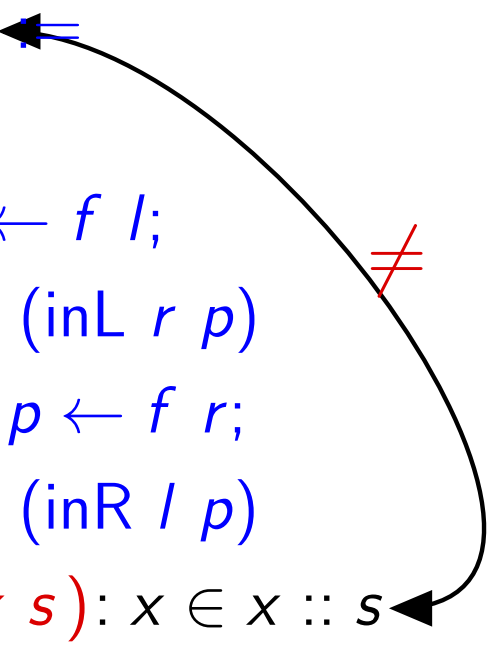
```
(1)   Definition search (x : A) :=
(2)       mfix f (s : list A) : M (x ∈ s) :=
(3)       mmatch s with
(4)       | [l r] append l r ⇒ mtry  p ← f l;
(5)                                  ret (inL r p)
(6)                              with _ ⇒ p ← f r;
(7)                                  ret (inR l p)
(8)       | [s'] (x :: s') ⇒ ret (in_head x s)
(9)       | [y s'] (y :: s') ⇒ p ← f s'; ret (in_tail y _ _ p)
(10)      | _ ⇒ raise NotFound
(11)      end.
```

search in Mtac

Typed!

Type checker

```
(1)  Definition search (x : A) :=
(2)      mfix f (s : list A) : M (x ∈ s) :=
(3)      mmatch s with
(4)      | [l r] append l r ⇒ mtry  p ← f l;
(5)                                 ret (inL r p)
(6)                            with _ ⇒ p ← f r;
(7)                                 ret (inR l p)
(8)      | [s'] (x :: s') ⇒ ret (in_head x s ): x ∈ x :: s
(9)      | [y s'] (y :: s') ⇒ p ← f s'; ret (in_tail y _ _ p)
(10)     | _ ⇒ raise NotFound
(11)     end.
```



search in Mtac

Typed!

Type inference

```
(1)   Definition search (x : A) :=
(2)       mfix f (s : list A) : M (x ∈ s) :=
(3)       mmatch s with
(4)       | [l r] append l r ⇒ mtry  p ← f l;
(5)                                  ret (inL r p)
(6)                              with _ ⇒ p ← f r;
(7)                                  ret (inR l p)
(8)       | [s'] (x :: s') ⇒ ret (in_head _ _)
(9)       | [y s'] (y :: s') ⇒ p ← f s'; ret (in_tail y _ _ p)
(10)      | _ ⇒ raise NotFound
(11)      end.
```

search in Mtac

Typed! Interactive programming

```
(1)  Definition search (x : A) :=
(2)      mfix f (s : list A) : M (x ∈ s) :=
(3)      mmatch s with
(4)      | [l r] append l r ⇒ mtry  p ← f l;
(5)                                 ret (inL r p)
(6)                            with _ ⇒ p ← f r;
(7)                                 ret (inR l p)
(8)      | [s'] (x :: s') ⇒ ret (in_head _ _)
(9)      | [y s'] (y :: s') ⇒ p ← f s'; ret (in_tail y _ _ p)
(10)     | _ ⇒ raise NotFound
(11)     end.
```

search in Mtac

Typed!

Interactive programming

```
(1)  Program Definition search (x : A) :=
(2)      mfix f (s : list A) : M (x ∈ s) :=
(3)      mmatch s with
(4)      | [l r] append l r ⇒ mtry  p ← f l;
(5)                                 ret _
(6)                            with _ ⇒ p ← f r;
(7)                                 ret _
(8)      | [s'] (x :: s') ⇒ ret (in_head _ _)
(9)      | [y s'] (y :: s') ⇒ p ← f s'; ret (in_tail y _ _ p)
(10)     | _ ⇒ raise NotFound
(11)     end.
```

A simple problem in Coq, with Mtac

$$\frac{x\ y\ z : A \qquad s : \mathsf{list}\ A}{x \in \mathsf{append}\ s\ (z :: x :: y :: [])}$$

Proof.

A simple problem in Coq, with Mtac

$$\frac{x\ y\ z : A \qquad s : \text{list } A}{x \in \text{append } s\ (z :: x :: y :: [])}$$

```
Proof.
  apply: run (search _ _).
Qed.
```

A simple problem in Coq, with Mtac

$$\frac{x\ y\ z : A \qquad s : \mathsf{list}\ A}{x \in \mathsf{append}\ s\ (z :: y :: x :: [])}$$

```
Proof.
  apply: run (search _ _).
Qed.
```

A simple problem in Coq, with Mtac

$$\frac{x\ y\ z : A \qquad s : \mathsf{list}\ A}{x \in \mathsf{append}\ s\ (z :: y :: x :: [])}$$

```
Proof.
  apply: run (search _ _).
Qed.
```

run is a first-class citizen

A simple problem in Coq, with Mtac

$$\frac{x\ y\ z : A \qquad s : \mathsf{list}\ A}{x \in \mathsf{append}\ s\ (z :: y :: x :: [])}$$

```
Proof.
  apply: run (search _ _).
Qed.
```

run is a first-class citizen

With lemma $L : a \in s \to P\ a$

A simple problem in Coq, with Mtac

$$\frac{x\ y\ z : A \qquad s : \text{list } A}{x \in \text{append } s\ (z :: y :: x :: [])}$$

Proof.
  apply: **run** (search _ _).
Qed.

run is a first-class citizen

With lemma $L : a \in s \to P\ a$
to solve $P\ x$ we can apply L (**run** (search x _)).

Use Monads

1. Encapsulate tactical effects in a monad.

   Give tactical effects a Coq type M *A*.

2. Execute (**run**) tactics at type inference.

} Mtac

Mtac: tactic language

Key idea #1:

- Put tactical effects in a monad M.

Mtac: tactic language

Key idea #1:

- Put tactical effects in a monad M.

Add typechecker.

Mtac: tactic language

Key idea #1:

- Put tactical effects in a monad M.

~~Add typechecker.~~ Coq is Mtac's typechecker!

Mtac: tactic language

Key idea #1:

- Put tactical effects in a monad M.

~~Add typechecker.~~ Coq is Mtac's typechecker!

Inductive M : Type → Type :=

| ret : $A \to$ M A

| bind : M $A \to (A \to$ M $B) \to$ M B

| mtry : M $A \to$ (Exception → M $A) \to$ M A

| raise : Exception → M A

| mfix : $((\forall x : A.$ M $(B\ x)) \to (\forall x : A.$ M $(B\ x))) \to \forall x : A.$ M $(B\ x)$

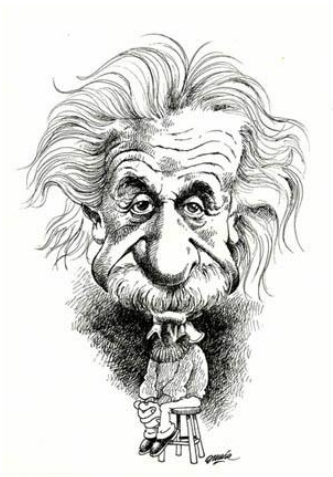
...

Mtac: tactic execution

Key idea #2:

- Execute (**run**) tactics at type inference.

Proof dev



Type inference

Kernel

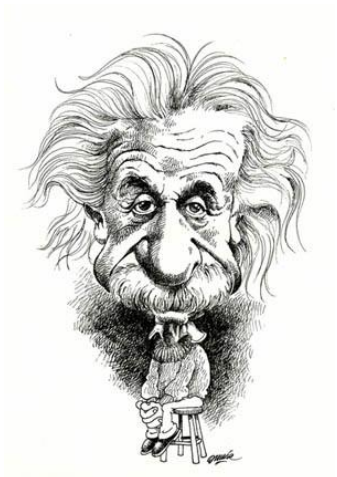
Mtac: tactic execution

Key idea #2:

- Execute (**run**) tactics at type inference.

$(\lambda x.\ x)\ 1$

Proof dev



Type inference

Kernel

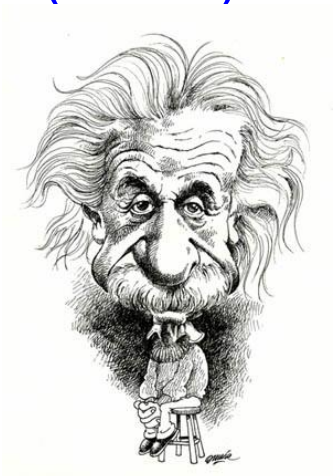
Mtac: tactic execution

Key idea #2:

- Execute (**run**) tactics at type inference.

$(\lambda x.\ x)\ 1 \longrightarrow (\lambda x.\ x)\ 1$

Proof dev



Type inference

Kernel

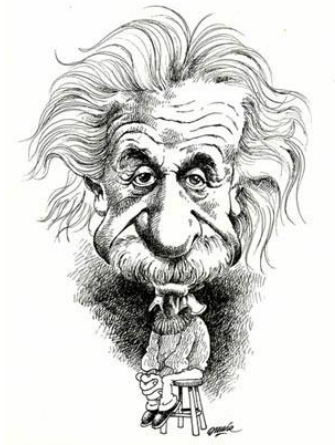
Mtac: tactic execution

Key idea #2:

- Execute (**run**) tactics at type inference.

$(\lambda x.\ x)\ 1 \longrightarrow (\lambda x : \mathsf{nat}.\ x)\ 1$

Proof dev



Type inference

Kernel

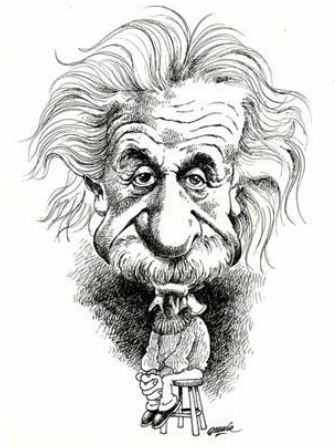
Mtac: tactic execution

Key idea #2:

- Execute (**run**) tactics at type inference.

$(\lambda x.\ x)\ 1 \longrightarrow (\lambda x : \mathsf{nat}.\ x)\ 1 \longrightarrow (\lambda x : \mathsf{nat}.\ x)\ 1$

Proof dev



Type inference

Kernel

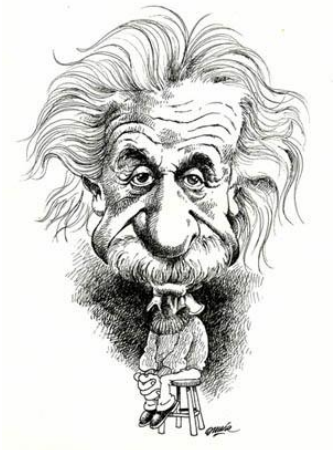
Mtac: tactic execution

Key idea #2:

- Execute (**run**) tactics at type inference.

$(\lambda x.\ x)\ 1 \longrightarrow (\lambda x : \mathsf{nat}.\ x)\ 1 \longrightarrow (\lambda x : \mathsf{nat}.\ x)\ 1$

Proof dev



Type inference

Kernel

Mtac: tactic execution

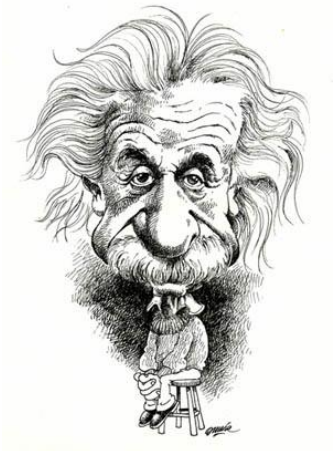
Key idea #2:

- Execute (**run**) tactics at type inference.

$$\Gamma \vdash e \hookrightarrow e' : A$$

$(\lambda x.\ x)\ 1 \longrightarrow (\lambda x : \mathsf{nat}.\ x)\ 1 \longrightarrow (\lambda x : \mathsf{nat}.\ x)\ 1$

Proof dev



Type inference

Kernel

Mtac: tactic execution

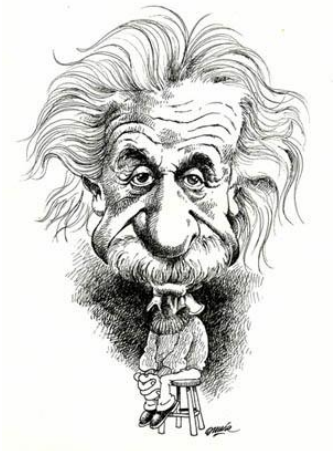
Key idea #2:

- Execute (**run**) tactics at type inference.

$$\Gamma \vdash e \hookrightarrow e' : A$$

$(\lambda x.\ x)\ 1 \longrightarrow (\lambda x : \mathsf{nat}.\ x)\ 1 \longrightarrow (\lambda x : \mathsf{nat}.\ x)\ 1$

Proof dev



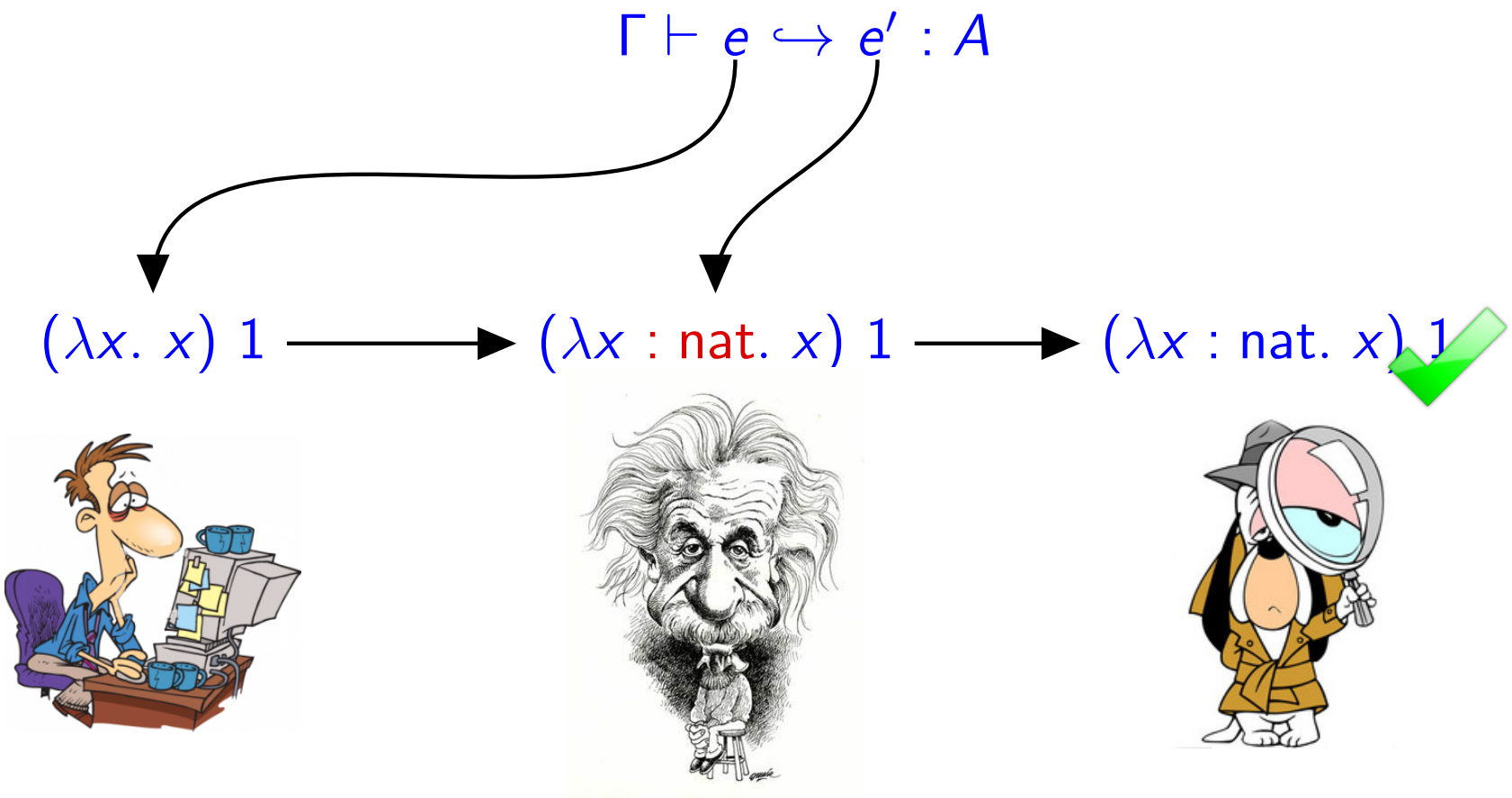
Type inference

Kernel

Mtac: tactic execution

Key idea #2:

- Execute (**run**) tactics at type inference.



Mtac: tactic execution

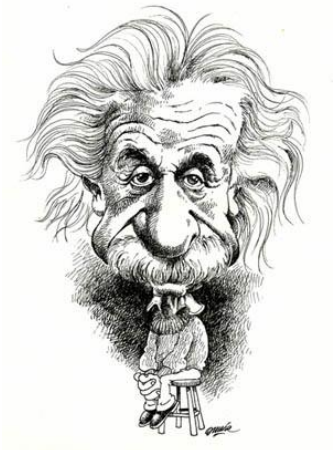
Key idea #2:

- Execute (**run**) tactics at type inference.

$$\frac{\Gamma \vdash t \hookrightarrow t' : \mathsf{M}\ A \qquad \Gamma \vdash t' \rightsquigarrow^* \mathbf{ret}\ e}{\Gamma \vdash \mathbf{run}\ t \hookrightarrow e : A}$$

$(\lambda x.\ x)\ 1 \longrightarrow (\lambda x : \mathsf{nat}.\ x)\ 1 \longrightarrow (\lambda x : \mathsf{nat}.\ x)\ 1$

Proof dev



Type inference

Kernel

Mtac: tactic execution

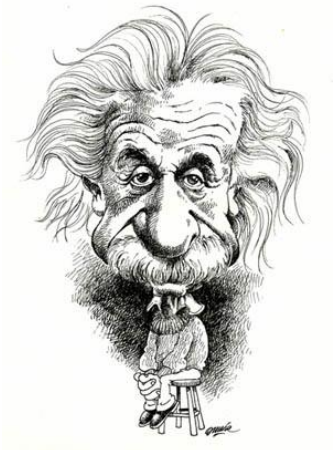
Key idea #2:

- Execute (**run**) tactics at type inference.

$$\frac{\Gamma \vdash t \hookrightarrow t' : \mathsf{M}\ A \qquad \Gamma \vdash t' \rightsquigarrow^* \mathbf{ret}\ e}{\Gamma \vdash \mathbf{run}\ t \hookrightarrow e : A}$$

$(\lambda x.\ x)\ 1 \longrightarrow (\lambda x : \mathsf{nat}.\ x)\ 1 \longrightarrow (\lambda x : \mathsf{nat}.\ x)\ 1$

Proof dev



Type inference

Kernel

Mtac: tactic execution

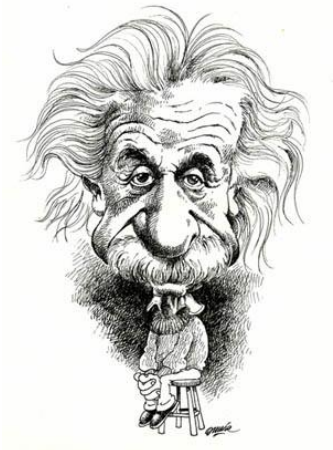
Key idea #2:

- Execute (**run**) tactics at type inference.

$$\frac{\Gamma \vdash t \hookrightarrow t' : \mathsf{M}\ A \qquad \Gamma \vdash t' \rightsquigarrow^* \mathbf{ret}\ e}{\Gamma \vdash \mathbf{run}\ t \hookrightarrow e : A}$$

$(\lambda x.\ x)\ 1 \longrightarrow (\lambda x : \mathsf{nat}.\ x)\ 1 \longrightarrow (\lambda x : \mathsf{nat}.\ x)\ 1$

Proof dev



Type inference

Kernel

Mtac: tactic execution

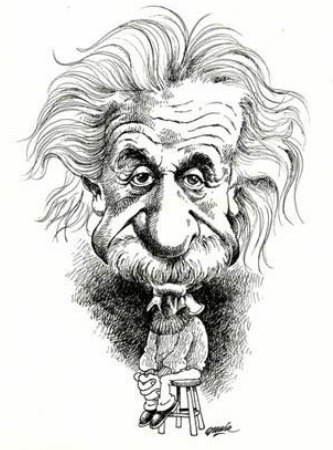
Key idea #2:

- Execute (**run**) tactics at type inference.

$$\frac{\Gamma \vdash t \hookrightarrow t' : \mathsf{M}\ A \qquad \Gamma \vdash t' \rightsquigarrow^* \mathbf{ret}\ e}{\Gamma \vdash \mathbf{run}\ t \hookrightarrow e : A}$$

$(\lambda x.\ x)\ 1 \longrightarrow (\lambda x : \mathsf{nat}.\ x)\ 1 \longrightarrow (\lambda x : \mathsf{nat}.\ x)\ 1$

Proof dev



Type inference

Kernel

Mtac: tactic execution

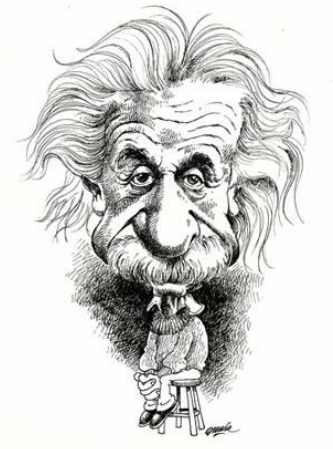
Key idea #2:

- Execute (**run**) tactics at type inference.

$$\frac{\Gamma \vdash t \hookrightarrow t' : \mathsf{M}\ A \qquad \Gamma \vdash t' \rightsquigarrow^* \mathbf{ret}\ e}{\Gamma \vdash \mathbf{run}\ t \hookrightarrow e : A}$$

$(\lambda x.\ x)\ 1 \longrightarrow (\lambda x : \mathsf{nat}.\ x)\ 1 \longrightarrow (\lambda x : \mathsf{nat}.\ x)\ 1$

Proof dev



Type inference

Kernel

Mtac: tactic execution

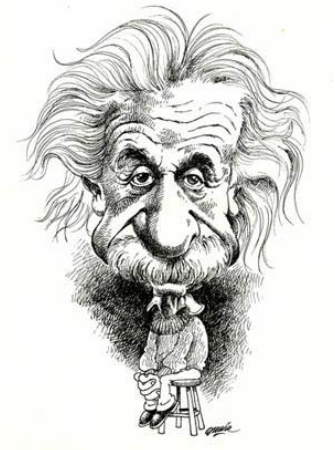
Key idea #2:

- Execute (**run**) tactics at type inference.

$$\frac{\Gamma \vdash t \hookrightarrow t' : \mathsf{M}\ A \qquad \Gamma \vdash t' \rightsquigarrow^* \mathbf{ret}\ e}{\Gamma \vdash \mathbf{run}\ t \hookrightarrow e : A}$$

$(\lambda x.\ x)\ 1 \longrightarrow (\lambda x : \mathsf{nat}.\ x)\ 1 \longrightarrow (\lambda x : \mathsf{nat}.\ x)\ 1$

Proof dev



Type inference

Kernel

Mtac: tactic execution

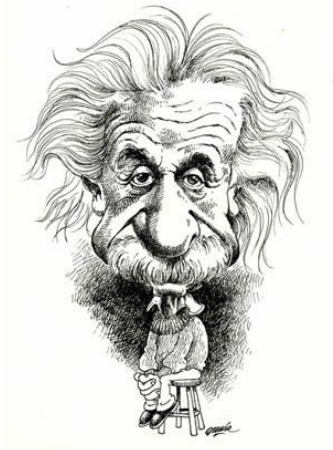
Key idea #2:

- Execute (**run**) tactics at type inference.

$$\frac{\Gamma \vdash t \hookrightarrow t' : \mathsf{M}\ A \qquad \Gamma \vdash t' \rightsquigarrow^* \mathbf{ret}\ e}{\Gamma \vdash \mathbf{run}\ t \hookrightarrow e : A}$$

$(\lambda x.\ x)\ 1 \longrightarrow (\lambda x : \mathsf{nat}.\ x)\ 1 \longrightarrow (\lambda x : \mathsf{nat}.\ x)\ 1$

Proof dev



Type inference

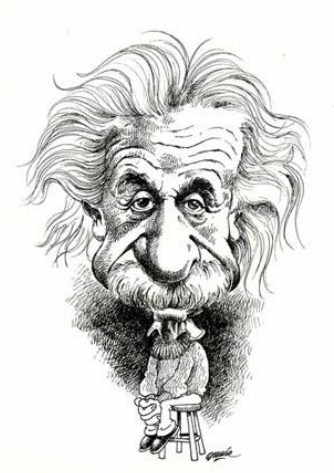
Kernel

Mtac: tactic execution

Key idea #2:

- Execute (**run**) tactics at type inference.

$$\frac{\Gamma \vdash t \hookrightarrow t' : \mathsf{M}\ A \qquad \Gamma \vdash t' \rightsquigarrow^* \mathbf{ret}\ e}{\Gamma \vdash \mathbf{run}\ t \hookrightarrow e : A}$$



Type inference

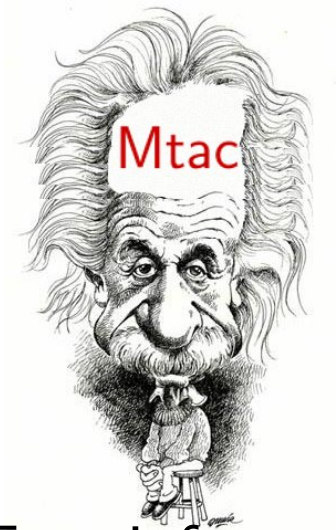
Kernel unmodified

Mtac: tactic execution

Key idea #2:

- Execute (**run**) tactics at type inference.

$$\frac{\Gamma \vdash t \hookrightarrow t' : \mathsf{M}\ A \qquad \Gamma \vdash t' \rightsquigarrow^* \mathbf{ret}\ e}{\Gamma \vdash \mathbf{run}\ t \hookrightarrow e : A}$$



Type inference

Kernel unmodified

Related work

(Some) other typed tactic languages:

- VeriML (Stampoulis & Shao '10)
  Beluga (Pientka '09)
  Delphin (Poswolsky & Schürmann '08)

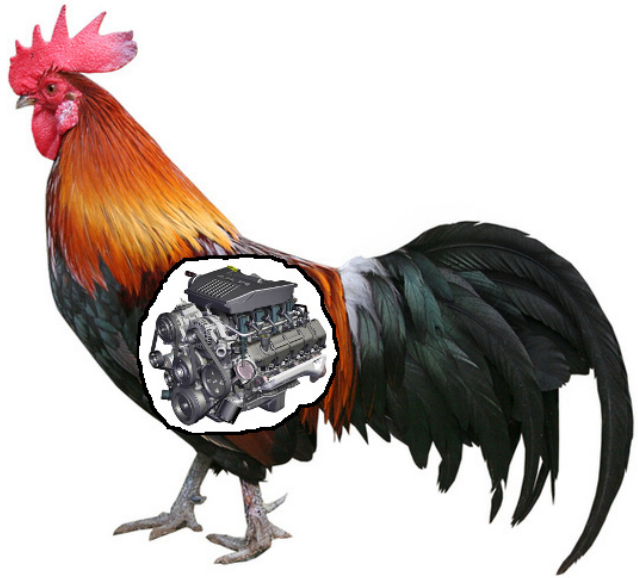
  Stronger typechecker, difficult to incorporate into Coq.

- Lemma Overloading (Gonthier, Ziliani, *et al.*, '11)

  Logic programming style, convoluted semantics.

In the paper

- Binder manipulation.
- Examples.
- Formalization.
- Implementation details.
- Related work.



http://plv.mpi-sws.org/mtac/